University of Warwick
Department of Engineering

Application of Definitive Scripts to Computer Aided Conceptual Design

Alan John Cartwright
MSc CEng MIMechE

A thesis submitted in compliance with the regulations for the award of the degree of Doctor of Philosophy at The University of Warwick. This work was carried out in the Department of Engineering under the supervision of Professor DJ Whitehouse

April 1994

Application of Definitive Scripts to Computer Aided Conceptual Design

Summary

The creative phases of design are based upon the human ability to conceptualise or abstract ideas from physical observations of the real world. That ability comes from experience, based on experiment: discerning patterns of behaviour in particular sets of observations. In this work it is shown that the process of identification, experiment and abstraction may be modelled accurately on a computer by definitive, or agent-oriented, programming, so forming a powerful aid to conceptual design

A new computer modelling language, called EdenLisp, has been developed by the author around Definitive Notations and interfaced to a commercial Computer Aided Design package. It provides a tool whereby computer models of systems can be originated that have state and on which state change can be made, not only by the designer but also by other autonomous agents of change.

Experiments with the language are described that show that scripts of definitions can have characteristics that permit the design to proceed as if there were an engineering prototype of the physical system being designed. The explicit representation of state at the lowest levels permits experimentation, observation of properties and addition of further observations.

The interactive construction of EdenLisp is analogous to the conceptual design process. It is used to illustrate and test design meta-theories for modelling conceptual design. It is shown to have potential for concurrent or multi-agent design, and is also an excellent vehicle for design education.

Contents

Acknowledgements

Interaction, intelligence and integration are siren words for Intelligent CAD systems. They also represent an accurate description of the way that the research group led by Meurig Beynon has helped to develop creative thinking and thoughtful insights that have contributed to this work. I owe an incalculable debt to Meurig and gladly acknowledge all his patience and support over six long years.

Supervision of prima donna academic staff is somewhat difficult, especially with an arcane subject, so I have appreciated Professor David Whitehouse's gentle encouragement and helpful guidance.

Any work of this magnitude takes a slice out of one's life, so I am most grateful for the support and encouragement of my wife Christine, and children Simon and Rebecca who have had their family life dominated by "definitive notations". I am thankful to God, the Great Designer, for the beauty of the world that is His design, and for His care and concern for His children.

Glossary

* indicates a term that is also in this glossary

Abstract Definitive Machine (ADM) A method of structuring a definitive program* into different scripts* each of which can be under the control of an agent; (agents may be acting concurrently). The ADM divides definitive statements into three

- those that the agent can unconditionally redefine*,
- those that the agent can redefine conditionally,
- those on which the agent imposes conditions on the ability of other agents to redefine.

CAD and **CAD/CAM** Computer Aided Design / Computer Aided Manufacture. *Design* is often a misnomer in commercial CAD systems that are actually *Draughting* systems. CAM usually implies a system for producing Numerical Control (NC) data to drive machine tools.

CADNO A Definitive Notation for graphics, like DoNaLD* but extended to 3-D.

Chunk A unit of integrated knowledge in long term memory. See also Percept*

Complex, Frame Terms used in this thesis for collections of labels arranged in ways that resemble topological nets and graphs. Nodes and the edges connecting those nodes are associated respectively with labels and the ways that the labels are grouped by means of parentheses.

Computational State In this work the *state of a computation* is what may be observed if a computation is suspended at any moment. The significant observations relate to the intention of the program that is running. A static state or statelessness refers to the condition where no observations can be made that were not preconceived by the programmer.

Connectivity The ability of labels to be connected by means of "edges" like nodes in a topological graph. Edges are not geometrically defined; they simply indicate connection. See also complex*.

Content The content of a symbol is whatever may be suggested by that symbol by any person looking at it. Content is always greater than that for which a symbol is used.

Declarative Programming A style of programming where statements attempt to express the desired outcome of the program in terms of "what is" knowledge rather than the "how to" knowledge of procedural programming*. Example are functional languages (Miranda, Lisp) and relational languages (Prolog).

Declarative Knowledge Knowledge about facts.

Definitive Programming (Definitive Notation) A method of programming where program statements take the form of definitions that are to be interpreted in their entirety without regard to their order. When a program, otherwise called a script* of definitions, is input to an Evaluator all the definitions are scanned and evaluated in the manner of a spreadsheet. Addition of further definitions, or redefinition of existing definition immediately trigger the evaluator; the whole script is scanned and re-evaluated as necessary.

Definitions take the form `variable = statement`
where `variables` are conventional computer variables and
`statements` may be values, formulae or functions, or calls to procedures.

DoNaLD is the Definitive Notation for Line Drawing, a notation that permits line based graphics. It translates a script* into EDEN* notation that must then be evaluated by EDEN. CADNO* and SCOUT* are other notations that translate into EDEN.

DXF, IGES, STEP, XBF Intermediate (neutral) codes used to transfer graphical data between different systems, especially between CAD* systems.

EDEN The Evaluator for DEfinitive Notations. A script* submitted to EDEN is evaluated in the form of a generalised spreadsheet.

EdenLisp The definitive evaluator written in AutoLisp, a superset of Lisp, and interfaced with AutoCAD.

Form A symbol used in mathematics or computer programming has an intended meaning in the context where it is used. This is the form of the symbol. Other meanings may be ascribed to the same symbol that were unintended and not in the scope of the application. The latter is the content*.

Frame See complex* or percept*.

GKS, PHIGS Programming software libraries for producing and displaying graphics.

Heuristic A rule for problem solving that is based on the semantic characteristics of the problem rather than its abstract characteristics. It is essentially a search based rather than an algorithmic rule.

ICAD Intelligent Computer Aided Design.
III-CAD The name used by the Dutch CAD research group for Intelligent, Interactive, Integrated CAD systems.

Instantiations are particular instances or examples of an abstract form that are worked out to a more concrete or observable form.

Instantiations are particular instances or examples of an abstract form that are worked out to a more concrete or observable form.

Latent states A definitive script* exists in a particular state (the state of the dialogue*) after it has been evaluated. A change in any definition will change that state. The set of possible redefinitions is infinite but if related to the physical interpretation that set represents all possible or *latent* states of the model represented by the script.

Meta-theory The prefix meta implies a generalisation. A meta-theory is a theory about theories: principles that apply to all theories.

Monotonic reasoning means progressing incrementally from one step to the next by reference to the starting conditions only.

Non-monotonic reasoning is making only one logical step at a time, reflecting on the outcome on the basis of real world observation before proceeding. The result may be a revision of the starting conditions at each step.

OOP = Object Oriented Programming. A style of programming in which programming objects are created that have local state because they have **encapsulated** or hidden variables. Neither those variables nor their values are directly accessible by other objects or procedures. At its simplest such an object may consist of a single procedure. Inputs and outputs connected with the object are of a specific kind allowing standardised linking of objects. An example language is Smalltalk.

Percepts are consistent and lasting ways of grouping observations made by humans on the basis of behaviours that link those observations in ways that seem to be predictable. In the Artificial Intelligence world, percepts are sometimes called **frames.** Frames* are also used in a different way in this thesis.

Procedural Programming A method of computer programming by means of a recipe or set of instructions for obtaining an output from given inputs. Examples are Fortran, Pascal, C.

Procedural Knowledge Knowledge about procedures.

Prototype A prototype in an engineering context is a physical object that is manufactured as the first of the *complete* product of the design process. It is used to check the performance of the product against the specification.

In computer science a prototype is a computer model that is used to check *particular* aspects of the software.

In this work it is used more in the engineering sense.

Redefinition If a definition within a definitive script is revised and resubmitted to the Definitive Evaluator then the whole script is re-evaluated. Redefinition may

mean changing its value or totally re-writing it. Provided the redefinition is consistent with the declared types of the variables used any alteration is legal.

Rule-based Systems are often called Expert Systems. Input is referred to an *inference engine* a computer program that causes the input to be tested against the knowledge and rules in the data or knowledge base. The output is a set of hypotheses (preferably of size one) representing a "desirable" outcome.

SCOUT A definitive notation* for interfacing with the Graphical system under X-windows.

Script, (Definitive Script) A set of definitive* statements, roughly corresponding to a conventional computer program.

State "An object is said to have state if its behaviour is influenced by its history. We can characterise an object's state by state variables, which among them maintain enough information about history to determine the object's current behaviour." [Abelson *et al*, 1985].

State of the dialogue When using a definitive evaluator the script* is scanned and evaluated after each definition is entered. The display and/or the internal representation reflects the state reached after the last definition was entered. This is called the state of the dialogue.

SDRC I-DEAS A commercial CAD/CAM system for mechanical design - a suite of programs based around a geometrical solid modeller.

TIFF A bitmap coding for transferring graphics data between graphics programs.

Virtual prototype A term used to describe the analogy between a physical prototype* and the computational model of the intended engineering design, that model consisting of a set of definitive scripts*, related according to the rules of the ADM*.

1

A Modelling Method for Experiments in Design

1.1 Introduction: Design as an experimental activity

Design presupposes a designer. That argument for a Creator is philosophically attractive. It is also an argument that suggests that engineering design cannot be automated to the exclusion of the human designer.

The link between design and designer is *choice*. Given the myriad possibilities of building the desired product from the synthesis of ideas with intransigent physical reality, decisions have to be made. The conceptual stage of the design process cannot be reduced to methodical procedures, despite some attempts to generalise methods that were devised to deal with certain routine tasks. One must carefully distinguish routine tasks from non-routine. Failure to do so leads to the belief that it is possible to automate design totally. Roth, following the German tradition for methodical procedures, thought that within ten years, methodical design would dominate and pure intuition become the exception. [Roth, 1981]. His interest lay in automating tasks that are well de-fined and for which solutions may be formulated that enable speedy choice of a relatively few options. But, as David Pye of the Royal College of Art put it

> "It is said that by the aid of computers we can arrive at the correct solution ... with certainty. They are clever, these computers! They are going to show us the cheaper answer. But if they think their clients are going to be satisfied with that, they are not so clever as they think." [Pye 1983].

Contrasted with the 'methodical' school is recent American research on design systems. At Carnegie and Stanford "non-routine" tasks within the design process

are defined by Piela *et al*, as "ill-defined, because they involve incomplete task descriptions and non-deterministic solution paths." They also support the notion that human interaction is essential to design: "Because non-routine design requires on-going human intervention, facilitating designers in this work will depend crucially on how well systems support use" [Piela, 1992]. Piela's work is significant in highlighting the experimental nature of conceptualisation. In the days before computer models replaced the difficult and expensive process of physical prototyping it was well understood that the nature of the physical world is such that frequently more could be extracted from a physical model than might be anticipated when the model is conceived. The act of building the model may reveal hidden manufacturing or feasibility problems, or simpler ways of achieving the same functionality.

In this work a design philosophy is developed that shows that designs are built on the designer's perception and experience of the physical universe. By experiment those perceptions are changed and improved. Thus they are based upon observation, both personally acquired and passed on.

The number of observations of the physical universe that can be made is infinite, both in category and within category. Thus one speaks of "design space", or "search space", by analogy with linear programming, (see, *e.g.* [Starkey, 1992]) by which attempts are made by specification to make limits within which the search for a feasible solution can be investigated in the design time allowed. In pre-computer days the design space was often narrowed to the extent of allowing only one solution: the first one that could be done that looked feasible. Despite that the environment for the search was still experimental. Indeed the word "search" well illustrates the experimental flavour of design. The search may not be done according to a preconceived specification but may be modified in the light of each finding. As the observation base grows so the problem definition evolves and with it the product. A client comes to a designer wanting a wall for his garden and goes away happy with a hawthorn hedge because the designer probed the reasons for the wall rather than merely discussing the type of wall.

A further strand to experimental design is developed in this work, namely the idea of *agents* in design. Design is normally the function of more than one person or agent, each interacting with the design, and often independently. An understanding of the way that agents operate is essential to a "theory of interaction" in design. The agent may be a person, making inspections or changes in the state of a design,

or the agency may be built into the design itself. For example a simple 4-bar linkage may have alterations made to its geometry by the designer: however a change in the angle of the driving lever will cause the whole mechanism to change its state by virtue of the connectivities of the bars. Agents often have choices: a warning light can be disregarded, constraints can be violated or redefined. Indeed these choices often provide the way that design progresses. Ways need to be found for coping with that level of interaction. In particular the issues of simultaneous and conflicting interactions are crucial in cases where there is a number of independent agents.

This thesis is concerned with the experimental aspects of conceptual design and how to support observation and experiment on computer-based systems. It is argued that any attempt to create such design support systems must provide an environment with the following properties.

- It is able to record observations of the real world.
- Computer models can be set up with attributes of physical prototypes, *i.e.* they have states that can be manipulated in infinitely many ways.
- It puts no constraint upon invention, particularly in the early stages.
- It allows interaction with the design by different agents acting concurrently.

On the other hand, in experimental investigations on how designers do design, it was observed that there is constant alternation between searching for a solution and fixing a solution; fixing the solution proceeds rapidly from success in the search for a solution [Ehrlenspiel & Dylla, 1989]. Access to standardised or automated procedures becomes advantageous at the point of fixing solutions, so methodical approaches need to be integrated. The assertion that the attempt to automate design is futile does not invalidate advances in methodical design. The two areas are complementary. The scope of the thesis is the conceptual stage of design.

1.2 A New Modelling Method

Interaction is fundamental to most CAD systems. It is that which has made them useful in conceptual design. However that is one of the few points in their favour. The modelling of a design on conventional CAD leaves the interpretation of the model firmly with the user. Modelling a design with real 'meaning' and enabling the design process are extremely difficult to do and are a major area of current research. Recently developed design support systems point up the importance of

appropriate computer-languages for design tasks, but also expose deficiencies in programming methods. For example, the modelling of state and state change is an important aspect of design but a traditional computer program *describes* state rather than itself being thought of as having state.

In this work a new programming language has been developed by the author to support the central thesis that computer models to help design should be situated in reality rather than being abstracted away from reality. Current programming techniques rely upon pre-selection of a set of observations with their own logical integrity in order to create modelling methods to aid the designer. Those methods mean that the designer effectively enters a logically predictable world where autonomous agents (such as, for example, those that are familiarised by the phrase "Murphy's Law") do not operate. In contrast, I created the language EdenLisp to permit a more realistic design world. It is developed from a programming method (paradigm) called the Definitive Programming framework, first developed at Warwick University. Using this method, it is possible to make a computational model as a metaphor of physical reality that allows one to examine that real world as one might by experiment and observation.

The definitive programming concept may be introduced by reference to one of the antecedent languages of EdenLisp developed by [Beynon & Yung, 1987] called EDEN.

> "The key idea behind definitive programming is the representation of computational state by a set of definitions of variables and of a transition between states by a set of redefinitions. A simple application of this principle underlies the spreadsheet. By way of illustration (when augmented by definitions of voltage *etc.*) the set of definitions
>
> ```
> resistance = resistance_of_lamp + cable_length * coeff_of_resistance
> current    = if switch_on then voltage / resistance else 0
> light_on   = switch_on and current >= threshold
> switch_on  = false
> ```
>
> can be interpreted as describing the state of a simple electrical circuit. In this context an appropriate transition might involve the redefinition
>
> ```
> switch_on   =   true."
> ```
>
> [Beynon, 1990]

Each new definition is interpreted in the light of previous definitions. If dependent variables are without current values the relationship remains unevaluated; only

when they all have values is the definition evaluated. If any variable acquires a new value at any time the definition is immediately re-evaluated. That point is significant. In conceptual design an object may be an instantiation of an idea defined initially in very abstract terms and only at the last "put into flesh". The definitive method allows such an approach. An object can be defined initially by formulae using labels that have no current values, nor even references to values. Nevertheless the formulae are binding and remain so when other definitions are made which build on them. In principle one could build up design specifications without needing to give values to important features, merely referring to them by labels: indeed one can construct a veritable algebra of labels in which ideas gradually attain substance.

As the definitive method was applied to design it is shown that it has the outstanding feature of being the means of creating computational objects that are each a kind of metaphor of a physical prototype. States defined by a set (or *script*) of definitions can be examined as one might physical states of real objects: no definitive statement is merely an adjunct to the computation. A redefinition is akin to changing a feature of a physical prototype or its state

For the interpretation and evaluation of definitive statements the techniques common to creating new languages are used. The original Evaluator of DEfinitive Notations (EDEN) was created by Beynon and Yung [Yung, 1987], and a number of derivatives of Eden were also developed. These definitive methods were the starting points for developing the main ideas described in this thesis, ideas that underlie the tools that are proposed for supporting design. For the engineering designer, a tool that uses Definitive methods would have little application without access to geometrical modelling: so *EdenLisp* was conceived. EdenLisp is a definitive CAD notation containing the basic Eden engine, but considerably enhanced with geometric extensions. As the name suggests, it is written in Lisp. It is specified by identifying an underlying algebra of sorts and operators to describe many different structural aspects of a geometric object.

Engineering design problems using EdenLisp show that the properties of the definitive method are indeed analogous to Engineering Design prototyping. Engineering models written in EdenLisp can be manipulated in a similar way to parametric models that sit on top of most existing CAD systems. The difference is that because it is a language approach there is no point at which the parametric

concept breaks down: it is definitive all the way down! Indeed it is conceivable that eventually all computation can be expressed using definitive scripts.

With that introduction the main "thesis" of this work may now be stated.

1.3 Thesis: Design Prototyping with Definitive Scripts

The process of engineering design, defined as a synthesis of physical observations and deductions with incremental refinement, may be modelled accurately on a computer by definitive, or agent-oriented, programming.

The author's Definitive Script interpreter EdenLisp provides a computer-based approach whereby models of systems can be originated that have state and on which state change can be made by independent agents

Computer models produced that way can have state that is characteristic of engineering prototype designs of physical systems. The explicit representation of state at the lowest levels permits experimentation, observation of properties and addition of further observations. Development of the design exploits interactions, constrains certain directions and allows concurrent agents of change.

The interactive construction of definitive scripts is analogous to the conceptual design process. It may be used to illustrate and test design meta-theories for modelling conceptual design. It is also an excellent vehicle for design education.

1.4 Overview

The ordering of the chapters following reflects the intertwining strands of design and computation. First, the experimental nature of conceptual design is considered, highlighting the modern requirement for rapid prototyping of products and hence the need for design product models that exhibit state and state change behaviour similar to physical prototypes. Trends in concurrent engineering and the growing interaction of non-engineering experts are examined. The aim is to identify the kind of computer-based tools that can cope with very different kinds of interaction on the same design.

Second, the novel idea of computation as experiment is introduced. Properties of older definition-based methods used in engineering are examined and the development of Definitive methods described and contrasted with them. Methods

for coping with multi-agent systems are still emerging but promising progress is described.

Definitive tools for graphical interaction are described next, defining the requirements for design in terms of abstract object definition. An algebra based upon these abstractions is developed out of which grew EdenLisp. The design philosophy and structure of the language of EdenLisp are described. The link with the CAD system AutoCAD® illustrates the flexibility of the method in accommodating traditional programming styles whilst exploiting the power of prototyping. In contrast experiments using definitive notations developed by others in the research group are reported to reinforce particular issues in prototyping and to show the generality of the method.

In the next two chapters the implementation of, and experiments in EdenLisp are described in detail. EdenLisp is implemented as a strongly typed language that is easily extended as new operators are introduced. The lexical analyser, parser and type checker are described, together with the evaluator. The environments for dealing with geometrical objects are then described. Finally programming examples are used to show the input style and possible user environments.

Design Management is very much the issue when there are design teams consisting of people with very different skills. The idea of definitive scripts is readily adapted to prototype definitive designs that have multi user, multi task design scenarios. The possibilities are explored in that chapter.

Applications of definitive methods in design education are very attractive. Parametric design, optimal design, animation and self-teaching are each illustrated with examples that show the power of the method

The work is completed with a discussion of the topics dealt with, evaluating the thesis from the perspective of likely developments and future research.

2

Design as Experiment

A significant feature of the definitive method of computer modelling is the experimental nature of the interactions made by a user. It is that which makes it potentially attractive to the conceptual designer. In this chapter the idea of design as experiment is developed in order to draw out particular conceptual differences between the design process and the computational process. The aim is to explore what is required in principle to aid the conceptual designer and so provide the underpinning reasons for developing the definitive tools for design

2.1 Design and the Designer

Anyone attempting to give computer-based aids to design had better try to understand design and the role of the designer. These are formidable tasks. Even their definitions are unclear and virtually all writers on the subject seem obliged to formulate their own. Contrast the following examples: the academic Michael French; the industrial practitioner, Coplin of Rolls Royce; and David Ullman "a designer all my life".

> "Design is all the processes of conception, invention, visualisation, calculation, marshalling, refinement and specifying details that determine the form of an engineering product." [French, 1985].

> "Engineering design is a detailed planning process concerned with defining the package of product and service that will fully satisfy the customer while at the same time satisfying the expectations of the shareholders of the producer." [Coplin 1987]

> "The only way to learn design is to do it. A design process that results in a quality product can be learned, provided there is sufficient ability and experience to generate ideas and enough experience and training to evaluate them." [Ullman, 1992]

In these definitions design indicates purposeful activity, directed towards an end. Indeed some prefer the narrower definition of 'product realisation' as more accurately defining the task of the engineering designer. The phrase draws attention to the distinction between the process of designing, and the product of the process, whereas 'design' is used to describe both activity and result. Both French and Coplin imply that the process of design may be identified independently of the designer. Ullman is more sanguine. He sees all three, the designer, the design process and the resulting product, as closely bound up with one another.

We need to examine the relationship among these three: the product, the design process and the designer. Is the designer central to design? In an automated design system what is the role of the designer? Can we automate the process of design to the extent of excluding the human designer, or must the computer be a design support system with the designer in charge? If the computer and the designer are agents in the design process, how do they cooperate, and what if there are many agents, as in a concurrent design approach?

We consider first the relationship between the designer and the designed product. The ability to design is often thought of as arising from our human propensity to see order and pattern in the universe. Conversely where we perceive pattern we tend to regard it as somehow pointing to a designer. Indeed observing design and pattern in the universe as a whole, people from engineers to artists, physicists to theologians, speak of a "designer universe". Paul Davies, a professor of mathematical physics, writes the following.

> "The natural world is not just any old concoction of entities and forces, but a marvellously ingenious and unified mathematical scheme. Note, words like "ingenious" and "clever" are undeniably human qualities, yet one cannot help attributing them to nature too.
>
> According to Christian tradition, the deeper explanation is that God has designed nature with considerable ingenuity and skill ... so as to permit life and consciousness to emerge. Our own existence in the universe formed a central part of God's plan." [Davies, 1992]

The significant feature here appears to be the total involvement of ourselves with the universe. It seems that any definition of design that fails to include the agents of design is incomplete.

At first sight we may consider that demanding such a strong link between the design and the designer is unnecessarily restrictive. Even given a designer, may there not be a distancing of designer and product? Cannot parts of the design process be independent of the designer? For example, the more routine components of French's description of the design process are amenable to standardised approaches, even where the task is new. The correct specification of such "standard" problems quickly yields answers, as Roth's recent work shows [Roth, 1989]. Within a narrower definition of design, or perhaps downstream of the conceptual stage there will indeed be standard components and processes, but that begs the question of the need for a designer since those standard components must at one time have been conceived. Producing variants of existing designs is an extremely useful annex to the design process but can it claim to be 'design'?

The question centres on conception, invention and visualisation. If those could be independent of the designer then it should be possible to construct a computational system in which designs could be inferred. In that case a computational model, consisting essentially of symbols could create truly novel designs, *i.e.* designs that make new relationships and new patterns on observable data. There is little doubt that certain new relationships can be formed, but genuinely new pattern recognition requires a data base that is considerably larger than the biggest data system yet devised. The reason for that is in the contrast between the form of symbols and their 'content'. Cantwell Smith, in a paper entitled 'Two lessons from logic' [Smith, 1987], argues the 'irreducibility of content to form'. By this he means that there are two factors to be considered in any symbol system. The first factor involves the shapes of symbols, how they can be put together and the behaviour or operations defined over the systems. That would typically relate to operations in a computer based system. The second factor has to do with what symbols mean, what they are about: in other words their content.

The problem that Smith identifies is that content is not intrinsic to the symbols. He gives the example of the word PLANE_{17}, a symbol that deals with the flight of a particular aircraft. No examination of the symbol within the air-traffic control system will reveal which aircraft in the air is referred to. To get that information one needs to look outside the system, to see how it is connected with the real world. The content of symbols outstrips the local assignment in a symbol system. Worse, there is a whole world of reference bound up with symbols that can hardly be comprehended. What, for example, does one make of the following?

"Why does this sentence remind me of Agatha Christie?"

There is no reason for that sentence to bring to mind the whole fictional world of Agatha Christie, yet it does! 'Reference outstrips causality' is how Smith defines it.

Reference of that sort, or 'content' is a human thing. It depends upon memories, experiences, knowledge, understanding. These may be buried in the sub-consciousness, but ready to be triggered at a word, a smell, a picture, a sound. The "content" of a symbol may be huge, spreading its web of connections ever more tenuously to include perhaps one's whole personal experience. The connection may be rational, it may be intuitive. It will also differ, depending upon the person.

Yet, it might still be argued, a 'content' may be large, but it can be regarded as finite in relation to a circumscribed model addressing a particular function. Is it not possible to construct a symbolic system that holds all that one might need for representing that model so that it has a one-to-one correspondence with the real world? The difficulty is in deciding whether one has 'all' that one needs. One cannot inspect that possibility, for one then comes up against what is called in computability theory the Halting Problem.

> "It is the question of whether there exists any computer program that can inspect other computer programs before they run and reliably predict whether or not they will go into infinite loops. The answer turns out to be "Definitely not" ... With that result there is no finite mechanism that can detect all patterns, patterns of patterns, patterns of patterns of patterns...." *On the seeming Paradox of Mechanising Creativity* [Hofstadter, 1979]

Any given logical system has to have a system outside; and that 'outside system' has to have a system outside it, and so on, to infinity. Whatever system one might construct to model the design process, its form must be limited - its content always connected to the real world by us as humans. It seems that 'agents are what matter' [Smith, *op cit.*].

If we wish to design a new object, the information enabling that object to be truly novel is likely to reside more in the mind and experience of the designer than in the forms that can be recorded in an automated system. Humans can make connections from among infinities of possibilities in apparently irrelevant experiences far beyond that which it would be reasonable to store in a comparatively specialised system for design support. We need to examine the ideas of symbol and content in more detail to see how those connections may be made.

2.2 Symbol and Content

The richness of 'content' in a symbol is illustrated in the widespread use of 2D CAD systems. While a rectangle might represent itself, it may be perceived, depending on the context, as a shaft, a box, a window, a zero symbol, a button or an information block. In other contexts it may be a kitchen cupboard, a tile, a sink; in yet another: a house, hotel or shop in plan view. The content is limited only by the imagination. A 2D CAD system permits the user to use imagination to its fullest: content is added by the user. The difficulty then is communicating the content to others: a difficulty that becomes acute in a multi-agent or con-current design. The traditional way out has been to have agreed conventions for interpretation such as BS308 Drawing Office Practice. It is well known that an engineering drawing conveys much more than that which is actually on the drawing. For example the presence of a fine-machining symbol on a dimension implies that the particular shape feature dimensioned is highly significant to the function of that object and drawing attention to it in a way that merely obeying the symbol to the letter may not be adequate. Another example is the tolerance on a hole position. That is specified in a way that cannot ever be measured: the centre of the hole is machined away; all you have is the rather inaccurate form of the circumference from which to infer the probable centre.

Despite the richness of possible *content* in a symbol the *context* of a symbol may provide certain avenues for exploring it in a rational way. Notwithstanding the intuitive connections, rational or experiential connections between symbol and content can be made by listing rules and relationships. It is the enumeration of finite groups of possibilities that underlies the taxonomic and the methodical approaches to Design. Characteristic of that is the recent abundance of research into *feature editors*. In mechanical engineering applications, for example, feature editors have libraries of features, such as holes, bosses, slots and pockets, that the user simply calls and assembles on the screen. It is claimed that all geometrical modelling could be based on such taxonomies, since other attributes can be added to standard features as required, and products designed that way can be readily linked with manufacturing activities. (See for example [Gu, *et al*, 1989], [Krause, *et al*, 1989], [Mantyla, 1990]. The latter gives a useful summary of feature editors).

The reasoning behind the taxonomic approach appears to be that one can construct commonly agreed vocabularies for referencing the world, or parts of the world.

There appears to be an important concept here. Roger Trigg, a philosopher at Warwick University, writes:

> "Language must be understood to be about one world open to public inspection. Plato makes it clear that it is a precondition of language that the world has a certain stability. ... We must have the assurance that when we pick things out and draw other people's attention to them, they too can identify what we are referring to." [Trigg, 1973]

Trigg is appealing to the lowest common denominator of all languages: an understanding of the world that is common to all humankind. That understanding is in turn fed by a common set of basic conceptions of the world. So if one can isolate and classify such experiences there is a possibility that the taxonomic approach has some merit.

The problem of classifying these "basic experiences" and common language is that of disentangling the common from the other content. One only has to listen to politicians to see that the same words can denote very different things! And 'features' identified at one time suddenly invert and become the opposite (black on white rather than white on black for example).

It is thought by psychologists that conceptions of the world are formed by early childhood experiences, reinforced by experience. Those conceptions take the form of 'percepts', mental pictures of things and events that tie up with a world view. Every time a new experience is had, existing percepts are used to try to tie it in with the world as already understood. For example Yvonne Wærn illustrates the way a concept based around "an orange" can be understood in terms of percepts [Wærn, 1986]. *Fig 2.1* shows some of the percepts around that single object.

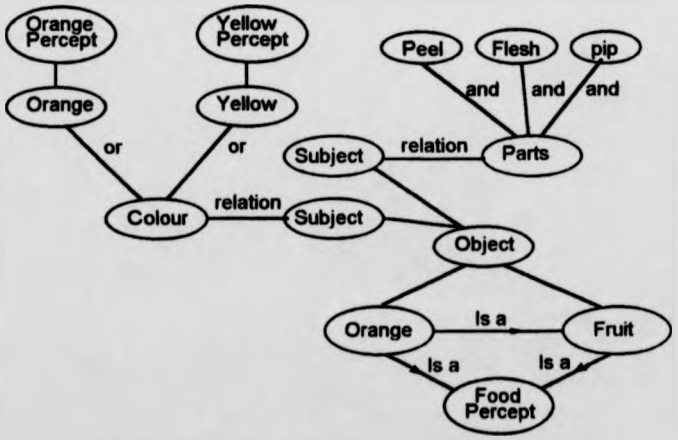


Fig 2.1 A Network of Percepts related to an Orange

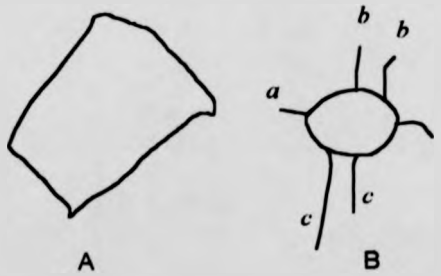
Percepts that are frequently used become reinforced and refined, and become "long-term memory"; less frequent or recent percepts are easily modified or displaced. The Swedish psychologist Professor Sandström made this observation of animals.

> 'The learning curve is not characterised by sudden leaps, revealing flashes of insight; learning takes place successively. When the animal has acquired a concept it becomes to a remarkable degree capable of altering the response pattern and adapting itself to the requirements of the situation.
>
> A so-called perceptive solution of a problem is here rather the result of extensive and versatile experience' [Sandström, 1968].

Sandström says that while this does not necessarily apply to human (as opposed to animal) insight, there does appear to be some evidence that percepts become the starting points for linking new experience.

Faced with a new experience new connections seem to be quickly established (first impressions) but later some regrouping of connections appears to take place as percepts get modified (second thoughts). A new experience is treated rather like the way the brain treats visual information from the eye. A literal data analysis of the retinal image yields a blind spot, but under normal circumstances the brain fills in extra information extrapolated from both visual and previously stored data. The new experience is like the blind spot, interpreted in the same way. New images, further experiences modify the extrapolated information and may later modify the longer term understanding. Donald Michie [Michie, 1986] mentions a startling illustration of this idea

> '[Fig 2.2] shows the result of asking a 3½ year old girl to copy a square. Her first attempt is on the left. Her second reproduced on the right, departs wildly from the first, and from anything that the ordinary onlooker might have expected her to do. The child explained that her descriptions indicated the corners, uprights and horizontals of the square respectively The phenomenon reveals the normally hidden operation of a particular way of compactly encoding percepts of external reality.'



Child's own commentary

a. "There for stiff things"
b. "For going up and down"
c. "These are the side bits"

Fig. 2.2 Drawing of Square by a 3½ year old. From [Michie, 1986].

Marvin Minsky, an Artificial Intelligence specialist at MIT, uses the word *frame* for what appears to be the same idea described by *percept*. He uses frames as a basis for understanding visual perception, mainly for natural language dialogues.

> 'When one encounters a new situation (or makes a substantial change in one's view of the present problem), one selects from memory a structure called a *frame*. This is a remembered framework to be adapted to fit reality by changing details as necessary.' [Minsky, 1981].

According to this theory, frames are formed in the brain with default assignments, the details of which may not be warranted by a particular situation but are useful generalisations sometimes by-passing 'logic'. Defaults are easily displaced by new items that fit better. So a hierarchy of frames is formed that becomes more stable with increasing experience, with interconnections that form a kind of relational database. This view appears to reinforce the idea that it may be possible to capture these percepts or frames in a knowledge base that can be used in forming a design support system that is independent of the user. Minsky's idea of frames (introduced in 1975) as a basis for Artificial Intelligence was very popular in the '70s and early '80s where, by analogy, a 'frame' was defined as a cluster of procedures and data referring to the same topic that could be moved around as one piece. [See discussion in Ritchie & Thompson, 1984]. Using that idea a number of attempts were made at 'frame languages' for natural language parsing, even the word 'percept' being used in [Sobolewski, 1988]. In the latter it is argued that functions and relationships do not exist; rather only attributes and values, as atomic conceptual primitives, and attribute paths (slots) that have the value of the most recent value. Percepts here are built up from the set of slots. The attempts have had mixed success.

We are still faced with several problems. Given a new situation, how do we locate a percept to represent it? Second, what is the content of that percept? Does there have to be a set of percepts that is common to everyone? Minsky */ibid./* says, 'We cannot begin any complete theory outside the context of some proposed global scheme for the organisation of knowledge in general'. But we have already seen that global schemes tend to be will-o'-the-wisp. If content is common to all humans then sharing, mimicking, learning are terms that relate only to acquiring that common experience. No, 'content' is peculiar to each person. Although there are shared experiences it is likely that those common experiences are linked to different percepts in different people. It may be possible to generate artificial percepts perhaps, with common factors that to some extent can be formalised. For

example many experiences are so common that they are rarely explicitly referred to; hence the famous conundrum 'Does a falling tree make a noise if there is no-one to hear it?', or the assumption that the hidden side of a cube is actually there and of an expected shape. Those assumed percepts could become micro-universes that have an agreed content. Even so that cannot mean that such percepts could be normative. World views do not remain unchanged in the light of increasing knowledge: compare the world views of Archimedes and Galileo, Newton and Einstein!

Content still outstrips symbol. If we could define a percept, its content would be impossible to limit. That is why 'natural language' is so often used in defining symbols. Readers of the symbol can interpret it with their own percepts. Symbols may be deliberately labelled in a system so that 'content' is appealed to that is not explicit in the system. Realism is implied, but is actually user defined. To refer back to the previous example, $PLANE_{17}$ is a symbol in a system. It could easily be labelled $ZIBKRLG_{17}$ and still do the same job, but without the same content implied by the word PLANE. In expert systems real-world vocabulary is commonly used although the system does not assign any 'meaning', merely the ability to access it according to some database management system. The advice "Sell your shirt to buy this computer", from an expert system for assessing computer systems, is simply an output pre-programmed to occur with certain input values. It could equally be programmed as "The probability of uptake for this computer is 0.95.", the result of a probability calculation from the same input values. Neither really expresses the real-world situation, where the expert might perhaps ask questions that depend as much on the customer's personality as the specified requirement, using assumed experiences of the client to guide the level of questioning. Again, agents and processing cannot be ignored.

2.3 Design and Observation

The content of an event or situation is an interpretation in the universe according to personal experience. In that sense significance is not innate - it is in the mind of the beholder. Minsky suggests the following process for dealing with new situations in terms of frames (with my numbering).

1. **EXPECTATION**: How to select an initial frame to meet some given conditions.
2. **ELABORATION**: How to select and assign subframes to represent additional details.

3. **ALTERATION**: How to find a frame to replace one that does not fit well enough.
4. **NOVELTY**: What to do if no acceptable frame can be found. Can we modify an old frame or must we build a new one?
5. **LEARNING**: What frames should be stored, or modified, as result of the experience? [Minsky, 1981, *op cit.*]

What is interesting is the kind of ordering process that is going on: very akin to the design process. Replace the word *frame* with *design* and the connection is apparent. Contrast that with a General Design Theory suggested by Tomiyama & Yoshikawa, and developed in [Veerkamp, 1993]. Their basic idea is given as follows. From the given functional specifications a candidate for the design solution is selected (*cf.* Minsky, step 1) and refined in a stepwise manner until a complete solution is obtained. (2 & 3) The process is evolutionary, transferring the design object from one state to another to give a more detailed description. (3 & 4) To evaluate the state of the design various interpretations need to be derived to see if specifications are met. (Step 5).

Comparing some of the presentations at a 1988 Design Theory Workshop conducted by Sandra Newsome, Spillers and Susan Finger, further formalisations appear. Spillers and Newsome point out that conceptual design contains a cognitive, heuristic component. On that basis many theories can be developed around the design of products, but they claim that what is needed is a kind of principle that would apply to all design theories. That 'meta-theory' about design has as its starting points

- a high level of ambiguity
- a partially ordered structure of ideas.

A partial ordering lists ideas with links that are hierarchical. Operations over a partial ordering are therefore those of a hierarchical tree structure, with weights to indicate relative importance. Typical operations are simple search: to calculate the role or effect of special components by examination of the linking elements of the tree structure; imposing a structure; and representation of the system by partially ordered subsets [Spillers & Newsome, 1989]

Orderings of that nature, whether regarded as frame or design, are pattern seeking and abstraction of important identifying entities that may in turn be decomposed into other entities. Those orderings will be based on factors that are part of a person's 'content' for the particular entities. What is more they must be based upon observation of the real world, since content is highly dependent upon observation. The question of which observations is perhaps the crux of the design-

designer problem. We have already seen that the number of possible observations is infinite and so content is infinitely variable. We have also seen that there are sets of observations that change very rarely (hence learned) or not at all (hence instinctive). Those settled observations are clearly the most easy to put into formal representations. Even so there remains the possibility that the representation may limit the implied content and so stultify creativity. What is required is that symbol and content be linked more directly by the user so that language and modelling are not treated differently.

One way that designers can get out of the difficulties of opinion and ambiguity is to make a physical prototype of the product. In [Pugh & Morley, 1988] the following conversation with an R&D director is recorded about the kinds of models used.

'Director So we would then make a feasibility, and what we mean there is that we've identified the broad product concept and approach. But now we've got to see in technical terms. Can it achieve cost objectives and time objectives, and can it meet marketing and performance requirements? For example, if we want the product to operate in a particular way, can the technology achieve it? So there must be compatibility in achieving the requirements.

Interviewer: You do more engineering there?

Director Yes, there's engineering at this stage. For example, the type of feasibility items that came out of this is some very crude model .. nothing necessarily to do with our business - it's simply got the right technology in it. We pull out modules and start writing software and doing some tests

Interviewer: It's literally a bit of kit lashed together?

Director Sometimes yes; the aim is to quickly demonstrate functional possibility in areas of high risk. We also prepare industrial design models - because we have to sell.

Interviewer So the bread-boards, lego models, prototypes, whatever you're doing here - you're learning from it to refine the spec.?

Director Yes.

Here the reason that the models are physical rather than computer based is clear, especially with the point about industrial design models. The latter points directly to the different set of understandings that a customer has compared with the designer; but both can inspect a model. Both can, to quote the fictional schoolboy, Billy Bunter's famous phrase, "jab a fat thumb into the works". That is because both share the same reality despite differing perceptions. That would be much more difficult with a computer model for the reasons already discussed.

The conclusion is that the design process is best served in the same way as the original perception or cognitive process is, by the designer having direct contact with the real world, being able to make observations experimentally. That way the 'content' or significance is made immediately relevant to the observer.

2.4 Models and Prototypes

'Experiments' in conceptual design may be thought of as the tests made by designers on possible solutions that they have constructed on the basis of experience, using the initial specification. But experiment may have a deeper sense: the casting around for the idea itself can be thought of as experiments on alternative realities. That spinning out of ideas has a similar motivation to that at Minsky's 'Expectation' stage of perception: trying to match the specification with a possible reality. In that respect the ability to try out things is important. As the new reality - the new design - begins to take shape then aspects of the problem become apparent. This is the process of partial ordering and breaking up of the problem into smaller entities. At that stage modelling becomes useful.

Modelling a problem is not normally an attempt to replicate the design - rather it is to simplify aspects of it for closer examination. Consider the following definition of 'model'.

> **"Model** */n/* **1.** A representation, usually on a smaller scale, of a device, structure, etc.; **2.** A representative form, style, or pattern."
> *(Collins Concise English Dictionary)*

While *representation* has the connotation of equivalence, it could alternatively suggest some selectivity. A model may have salient features that symbolise the represented design while other features of the model may be irrelevant or even misleading. (Materials used in a physical model are often highly inappropriate to what is modelled, chosen more for ease and speed of manufacture.) 'Smaller scale' may mean smaller in scope or content rather than size. The reduction in scope could be quite drastic, as in a mathematical model. In that case the model is at the extreme end of the modelling spectrum having virtually nothing innate in the symbolism other than a mapping of perceived relationships. Such symbolic models are deceptive as one can confound the model for the thing itself, or else think the model is actually better than reality. The latter is akin to the Platonic idea of Forms.

> "We distinguish between the many particular things that we call beautiful or good, and absolute beauty and goodness. Similarly with all other collections of

> things, we say there is corresponding to each set a single, unique Form that we call an "absolute" reality. .. And we say that the particular are objects of sight but not of intelligence, while the Forms are the objects of intelligence but not of sight." [Plato, *Republic*, VII,6]

That idea can lead us into real difficulty if we think of the model as somehow representing the Form rather than the substance. For example, mathematics itself has been regarded as a fundamental Form. Platonists have opined that mathematics is *discovered*, not invented. Galileo declared "The book of Nature is written in mathematical language". In that case which is the *real* "reality"? An abstraction of a *triangle* models but one aspect of three sticks fixed together at their ends, but we can think of the three sticks as a not only a representation of the abstraction but with an infinite other 'content'.

These are the hard questions of artificial intelligence, a major source of ideas for implementing designer support systems. Bound up with modelling in AI is the idea that 'modelling by Turing machines can be taken to be "free" in that the model of X is interchangeable with X itself' [Smith, *op cit.*]. That agrees with the Platonist but rather goes against all we have discussed up to now. If we accept the Platonist view we get stuck with a Universal Turing machine as a model of the universe itself, a suggestion refuted by the self-reference argument already rehearsed. Despite the philosophical problems of parallel universes and alternative realities the 'safest' approach is to take the universe as it appears and to make observations as near first-hand as possible. We have seen earlier that that is the conclusion taken as obvious by those nearest the problems of design.

The kind of reality one can have seems to be in the mind of the designer when sketching out ideas. One only has to watch a designer sketch a line, pause, delete the line and then redraw the line in the same place to realise a pattern of thought has occurred that says *this* line is quite different from *that* line and a whole 'content' is implicit in that one line. The line represents the *intention* of the designer. The thought is well explained by Takala.

> "In philosophical terms, the implicit requirements [of a design] are called *intensional* descriptions of the product, whereas explicit models of the product are *extensional*. Designing proceeds in two opposite directions: *top-down*, starting from implicit functional descriptions and synthesising an explicit model, and *bottom-up*, analysing and combining explicit realisations and trying to find useful implicit properties in them" [Takala, 1989] .

The ability to record or model intensional ideas is essential if we wish to construct support systems that help designers to design the way that they actually design. For despite many attempts to codify design methods the process is still implicit, cognitive and abstract; it is still perhaps best described by the design folio: a logbook of ideas, sketches, half thought out concepts, back of envelope calculations that follows the designer's progress towards solutions. That folio tends to reflect a 'three steps forward, two steps back' movement rather any kind of block diagram stages of progress beloved of text-book authors of design.

A considerable number of approaches have been tried to deal with intensional ideas. The simplest is not to attempt to record them, but to leave them entirely with the designer. As indicated earlier, the 2D draughting system is a suitable tool to do that and its continued popularity is witness to the success of that approach. Constraint based methods go to the opposite extreme, being pursued to give fully logical synthesis of the intention. [*e.g.* Chan & Paulson, 1987, El Dahshan & Barthes, 1989]. Although numeric *constraint solvers* are well suited to problems amenable to analytic formulations, they severely limit human intervention and often prove to be unnecessarily global. In an effort to model such human intervention the *expert system* approach is invoked so that constraints can be written as rules. Such systems are only currently able to cope with fully constrained problems and both approaches are computationally expensive. The latter two approaches can sometimes by found as tools within bigger packages. (*e.g.* Pro-Engineer)

Another approach is to encourage the designer to record what intentions are around a given line or geometrical object. This was in part the idea behind Object-Oriented Programming - recording the purpose as well as the geometry of parts. The computer model is built up from the descriptions given by the user, forming objects with 'meaning' in the real world that have real-world connections. The difficulties in principle of doing that are apparent from our discussion. Practical difficulties will be discussed in the next chapter. What is important here is the idea of modelling real life relationships explicitly when such models are intended to convey implicit or intensional ideas.

Finally we consider how intensional ideas are conveyed by the building of a physical prototype. With a physical realisation the experimental approach implicit in the sketched ideas can be extended by allowing observations to be made and experiments performed. A physical prototype is more than just a replica of the designer's thoughts; it embodies reality and so carries more 'content' than the

idea. It is that which gives the prototype its superiority over other modelling methods. Properties of the prototype remain to be discovered. Also amendments and additions can be made to the prototype using features not in the original specification. (The industrial designer's ideas of colour are difficult to conceptualise, but a prototype in a different environment might suggest a very different colour. Successive bridges on the M6 motorway were painted in different colours because the wife of the architect suggested it would reduce boredom of motorists, a factor quite remote from the function or form of the bridge in its own locale.)

2.5 Concurrent Design

Many tasks in design overlap or can be done concurrently. This was well understood in the West at one time but forgotten: probably due to the high degree of specialisation and differentiation in job functions. In recent times the automating of certain separate functions gave rise to 'islands of automation' that were difficult to connect together let alone permit concurrency.

The Japanese never had that particular problem since they were used to multi-disciplinary teams with very high levels of communication. The difficulties of that communication are identified by Daniel Whitney in his seminar on Japanese methodologies.

> 'Success at product-process integration requires identifying just what information the downstream process designers will need from the upstream product designers, and vice-versa. In the absence of a structure for this data exchange, integration degrades into arguments and confusion.
>
> The research community has barely recognised this issue. Potential approaches include information analysis of design processes, cost structure analysis of fabrication and assembly and modularization methods for products' [Whitney, 1992]

[Wilson & Greaves, 1989] discussed very much the same issues on a broader front, coining the phrase 'Forward Engineering' for the activity based upon early manufacturing involvement, the management of technological uncertainty, quality function deployment, design for manufacture, and assembly and process control. Forward Engineering demands cross-functional teams representing these specialisms as well as conventional engineering design. On that model many core people participate in the early stages of product design

Technical developments in CAD have reinforced the demand for wider participation. At one time, comprehension of technical drawings was a prerequisite for appreciating an engineering design. Even after the draughting process was automated, the interpretation of computer-aided drawings remained a task for the specialist. Only in recent years, when sophisticated computing resources for graphics have become more widely available and more powerful techniques for visualisation have been developed, has it become feasible to animate a design realistically at a very early stage. As a result, more people are now able to appreciate a designer's proposal; marketing mangers, visual designers, service personnel, accountants - even conservators - all wish to interact with the design and express their view of the product requirement. [Cartwright & Beynon, 1992].

The focus of concurrent engineering is on a pattern of interaction that is more complex than that of a single designer. Design activity corresponds to passing a consultative document around between human agents. In that context design activity passes through phases that are product-oriented: specifications and schemes are hard-copy outputs that become inputs for the next phase. Evaluation goes on at all phases of design often being iterative. Evaluation criteria must be available on the basis of incomplete designs and possibly also on incomplete data. It is essential therefore that the roles of the different agents are circumscribed so as to avoid the 'arguments and confusion' Whitney observed.

Given the incompleteness of partial designs it is essential that constraints on a design are seen as provisional. For the engineering designer the space in which the solution to a particular design problem is normally limited by functional constraints. The designer team will want a host of other constraints: cost, quality, manufacturing feasibility, appearance, etc. One way of enabling progress on all these fronts is first to identify areas of the design that are exclusive to each expert and then the areas of overlap and interfaces to other areas. Normally the expertise of the design manager will cope with that: the whole purpose of team briefings is to delineate those areas. But if we try to automate the procedure what may happen is the simultaneous developments of different models of the same product, with all the dangers that brings! A sketched design becomes the solid modelled shape that is tested for stressing at the same time as it is being knocked into a better aesthetic shape and the manufacturer is trying to put in draught angles for casting. That done as a sequence is bad enough but if a series of slightly different designs emerges from concurrent actions there is a recipe for discord. It is then easy to see that a single prototype lends itself better to group design. Something of that is

possible using communications on computer systems, but then constraint management has to be built in. The idea of experiment even for constraints is then worth considering. Constraints would become things to be tested, modified or over-ridden rather than being written in stone as the easy way to control interaction. So all constraints would need to be explicitly justifiable, whilst preventing the non-expert from making unsuitable changes to areas that are central to another's area.

The main idea of experimental constraints is that different agents acting on a design can feel at liberty to try out 'what-if?' scenarios, knowing the nature of the monitor placed on the constraint. A design support system based on that idea would mean the model being tested would be the same for all agents and the agent clashes would emerge in the setting up of appropriately actioned monitors.

In our argument for experiment in design we conclude the following.

1. *The designer is an integral part of the design*
2. *Content depends on the designer not the model*
3. *Content is filtered through percepts that help to organise the world into hierarchies, but these percepts are different for each individual.*
4. *Observation and experiment are fundamental to our organisation of the universe, even to reference itself*
5. *Models cannot stand in place of reality, although some standard frames can be codified for bottom up design*
6. *Supporting intent is best done nearest to reality - physical prototypes invite experiment*
7. *Concurrent engineering requires good communication between independent agents. It is best done where different agents have access to the same model.*

3

Computational Modelling for Design

In this chapter the problems of making computer models are discussed. The principal programming tools are reviewed in some detail in order to expose the difficulties of using object description, as opposed to representation of objects with state. Finally it is argued that the Definitive Method can be adapted to provide a fundamentally different approach that is inherently better suited to modelling the design process.

3.1. Object and Process Models for Design

3.11 Background

Early computer based tools for aiding design, many still in use, were developed around mathematical or logical descriptions of particular aspects of a product. 'Design' problems were then largely analytical, extracted from the design concepts and requiring only the right formulation for their solution. Interactive design was understood in terms of enabling designers to specify a series of analytical problems correctly, guiding the computer system through the various 'islands of automation'. Consider the following quotation from a CAD/CAM handbook of the early '80s.

> 'Interactive process design is an application of CAD/CAM to the primary manufacturing process. The designer can work with polymers, ceramics and metals in processes such as moulding, casting, extrusion, and drawing. This involves three principal elements: process physics, simulation and computer graphics.

> First a fundamental understanding of the process physics involved is essential. Process actions and reactions must be reduced to mathematical expressions.
>
> The second element, simulation, takes the mathematical representations of the component geometry or shape and combines them with the process physics in computer simulations. Interaction with the computer model allows ... optimisation, .. and parametric studies. ... A common database allows design of related tooling.
>
> Third is visualisation, allowing synthesis of information from complex pictures and so effecting vital decisions. ... Thus the designer can interactively conceptualise, design and manufacture with greater control.
>
> Analytical computer programs for stress, deformation, flow and heat transfer [need to be tailored] so that they can be linked to CAD/CAM systems for interactive process design' [Miller, *et al*, 1980]

The target alluded to in Miller's last paragraph is *integration*. And a difficult one it has proved to be. A number of mathematical models are made to solve different problems on the same design. These often give divergent partial descriptions of the design. Their contents may overlap but with different notations and mathematical foundations. Early computer models were initially associated primarily with functional properties of the design, but with progress in computer science and hardware, models have become more wide ranging, more sophisticated and more diverse *(table 3.1)*. Developments in geometrical modelling have brought pictorial description to the level where there is much talk of 'virtual realities' and indeed some of the graphics available now deserves that epithet. Nevertheless all these different models could be regarded as digital descriptions of parts of the same real-world objects. Integration has become a moving target.

Functional and Performance
Computational methods for solving physical & mathematical relationships
e.g. Finite Element Analysis
Relationships: intra and extra
Heuristics: problem solving, optimisation methods
Geometrical form
Graphical Processor a la Word Processor: 2D/3D draughting system
Geometrical model creation and manipulation: solid modeller
Geometrical descriptive models: APT, PADL2
Feature editors
Shape grammars
Appearance and Attributes
Computer graphics
Data bases

Table 3.1 Examples of Object Description Software

Consider for example the design of a single arm robot with upper and lower arm and central axis rotation. Here geometrical models are needed of the form of the components and assembly. Various analytical areas can be dealt with using rather unrelated models - kinematics, control system, finite element analysis. The transformation of one model to another is therefore difficult or impossible. In the thinking of the '80s, getting information from one computer program to another was a matter of having intermediate or common forms that would be in a neutral format to enable transfer.

Common or exchange formats operate at different levels. The lowest level is the bitmap: a list of the actual bits composing the pixels in a picture generated on a raster, carrying information about colour and intensity in one or two bytes per pixel. That format is expensive on memory. A simple compacting method is to replace identical pixels in a raster row by a single pixel reference plus a byte to list the number of repeats. Bitmaps are the exchange formats used for transferring pictures to word processors for example (*e.g.* TIFF). A more sophisticated data transfer technique is to code graphic elements to make up line drawings and, more recently, solid and boundary represented geometry. That was used in the data exchange formats such as Initial Graphics Exchange Specification (IGES), Experimental Boundary File, and proprietary AutoCAD format (DXF). A third level of commonalty is the actual software used to generate graphic and other elements. One way to do that is to have graphics libraries that cover common elements in a way that is independent of computer hardware. Examples of such software are the Graphics Kernel System and the Programmer's Hierarchical Graphics Standard. Despite valiant efforts to push these as international standards, they are unpopular with commercial vendors because of speed problems. IGES is the most popular for CAD although valiant attempts are being made towards an ISO standard in STEP. In other areas the bitmap variants and the developments in X-windows graphics both point to a growing popularity of object oriented program objects.

Intermediate forms remain essential in a market of so many different proprietary software houses. But they do not tackle the basic requirement of seamless integration. An obvious alternative is for one vendor to combine a complete set of CAD/CAM software such as that by Computer Vision, or by SDRC (I-DEAS). That is not always satisfactory because data transfer is still implicit even if it is not seen by the user. It is not integration: simply a common and better user interface for the separate packages lurking behind the screen.

But a more serious charge can be brought. Intermediate forms simply record computer based information, most of which is not actually part of the physical object but is needed to reconstruct the representation. Examination of an IGES file may disclose the current representation of the design - it gives few clues as to where the design came from, or the designer's plan for its development. Integration really requires a more fundamental approach, bringing computer-based intelligence into the Design System. That is what has prompted the international efforts in research into interactive, integrated and intelligent CAD systems (prompting the name 'III-CAD' of the Dutch group of Akman, ten Hagen and Veerkamp).

3.12 Intelligent Integration of Models

[Tomiyama, 1989*a*] discusses the principle of intelligent integration. He introduces the idea of ***meta-model*** by which is comprehended those properties of models that are independent of what is modelled. Defining a ***model*** as a ***theory-based*** set of descriptions about the object world, he suggests that ***modelling*** is the process whereby object facts are filtered by the theory to formulate a 'world that is complete in terms of the theory'. It is important therefore to identify what is common to different models of the same system. He reduces that to a single meta-modelling precept: that all models of a given 'world' should relate to one another. An intelligent system built on that precept means that all descriptions of the design product are interrelated and built on a single set of information.

The construction of models based on a common view of the physical world requires more of a "real-world view" of design. That is itself a matter for conceptual design, and one is faced with the issue of 'content' discussed in chapter 2. (It also presupposes that we know what the "real-world view" actually is - a problem recently being tackled in terms of what is known as naive physics [*c.f.* Akman & ten Hagen, 1989].) The problem to be programmed has to be refined such that it not only isolates those characteristics that are amenable to examination but also allows for alteration and development of the evolving design. In other words not only must the models describe the objects, the system must deal with the design process itself. Attempts to use computer models themselves as models of the design process or to provide a computer based design support system raise philosophical issues relating to 'modelling' modelling systems, or 'meta-modelling' in an even broader context than Tomiyama's. *Fig. 3.1*, (overleaf) picks up the robot example discussed above, but with some significant additions. 'Content' in the real world is larger than can ever be contained by any theory. Some possible items of

'content' are indicated in *fig. 3.1*. For example the problems arising from the breakage of a designed object are rarely considered at the design stage. (A classic example of that omission occurred on a machine tool gear-box designed with a shear-pin that was supposed to shear when the load was too great. Indeed it did, but the pieces of the pin fell into other moving elements of the gear-box with catastrophic results! Similarly a British Gas maintenance survey found 50% of call-outs were directly attributable to faults that could have been foreseen and dealt with at the design stage). Other designations on *fig. 3.1* relate to the lay person's mental pictures of robots, fed mainly by science fiction. Although such pictures seem inconsequential it is interesting that on a number of occasions fiction has influenced design.

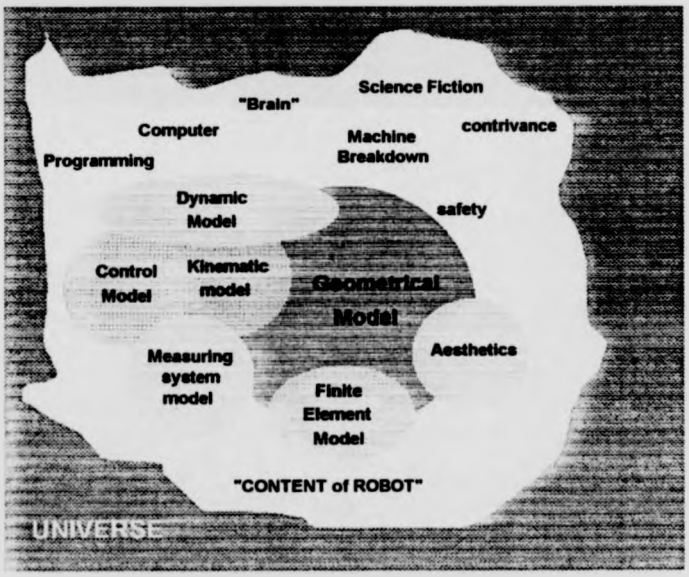


Fig 3.1 Aspects of Robot Design

One way out of the difficulty is to construct an artificial world, large but limited, basing meta-models on that universe of discourse, for example using a large number of shape features and a feature editor. Design support systems based on such ideas have been developed and form environments for design that enable track to be kept of both the process and the current stage of description of objects being designed. The discussion of those systems is deferred to chapter 4. Meanwhile we return to the fundamental problem of 'content'. Can one make a computer environment that contains observations about reality on which one can experiment, like a physical prototype? Alternatively, can one construct a computer model that

contains Platonic Forms is such a way that discoveries about the model will map into the physical world? At a practical level, in [Tomiyama and ten Hagen, 1987], the authors analyse the problem like this: an Intelligent CAD system should not only provide a medium for expressing, interrogating and modifying the current state of the model, it must represent and process *intensional* information (that which captures the functional requirements in the form of relations and other abstract notions). In the standard CAD model intensional information is not recorded, the designer may only represent the functional attributes of the product by selections from a pre-programmed set of preconceived operations. Progressing the design relies upon the designer being able to compare the current (*extensional*) computer model with the designer's intention and modifying the model until a satisfactory match is reached.

A confusion in modelling the modifying process has to do with the idea of 'state': a slippery relative of 'content' and equally difficult to capture in a computer system. It seems that rather than constructing massive edifices to model design and design state we need to go even further back than Tomiyama in our thinking. His valuable contribution points up the problem; his answer was to look to particular computer programming methods to construct a viable solution. Before we examine his solution it is necessary to look at computer programming generally to see how state and state change can be dealt with.

3.2 State and Computer Programming

According to [Abelson *et al*, 1985] state may be defined as follows.

> "An object is said to have state if its behaviour is influenced by its history. We can characterise an object's state by state variables, which among them maintain enough information about history to determine the object's current behaviour. "

Many Computer Scientists, particularly functional programmers, argue that state need not exist in computing. The contention is that computer models may be constructed using computer programming methods that are built on "logical" principles: principles tied ultimately to the Turing machine, the basis of all modern computers. Before examining computer programming languages, it is worth a digression to examine the point about the Turing machine and statelessness.

3.21 Automata

A Turing Machine is a finite state machine in which a transition prints a symbol on a tape. The tape head may move in either direction, allowing the machine to read and manipulate the input as many times as desired. A brief description of Turing Machines is quoted from [Sudkamp, 1988].

> 'The Turing Machine is abstractly a quintuple $M = (Q, \Sigma, \Gamma, \delta, q_0)$ where Q is the finite set of states, Γ is a finite set called the tape alphabet containing a special symbol B that represents a Blank, Σ is a subset of Γ without the blank called the input alphabet, δ is a partial function from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$ and $q_0 \in Q$ is a distinguished state called the start state. A machine configuration consists of the state, the tape, and the position of the tape head. At any step in the computation only a finite segment of the tape is non-blank. If a configuration is denoted by uq_ivB where uv is the string of items on the tape from left to right then uq_ivB indicates the machine is in state q_i scanning the first symbol of v. This representation of machine configurations can be used to trace the computations of a Turing Machine. [*op cit. section* 9.1]
>
> A sequence of elementary transitions represents a computation. Computations read and manipulate the symbols on the tape. The result of a computation can be defined in terms of the state in which computation terminates or the configuration of the tape at the end of the computation. [9.2]
>
> A Turing machine that computes a *function* has two distinguished states: initial state q_0 and final state q_f. A computation begins with a transition from q_0 that positions the head over the beginning of the input string. State q_0 is never re-entered. Its sole purpose is to initiate computation. All computations that terminate do so in q_f. Upon termination the value of the function is written to tape. [12.1]

Even from that very brief introduction it appears that if we can translate any problem into a finite set of symbols to form the input string to a Turing machine it is possible to formulate that problem with only one end state. Essentially that system is stateless, since only one outcome is allowed. Now a hypothesis known as the Church-Turing Thesis ensures the success of the Turing machine when it is modelling 'any problem that is computable' by coding from a finite set of symbols, whether those symbols be {0,1} or the ASCII set. So it is possible to formulate any problem based on mathematical foundations in terms of stateless abstractions.

We turn now to examine different types of programming languages and how they relate to state. *Fig 3.2* shows a very approximate taxonomy of the computer language types and examples of those types that are considered in this chapter. The

hierarchy is intended to indicate the level of complexity and the implementational dependencies of the languages. Generally the more complex languages may be implemented in either functional or procedural languages.

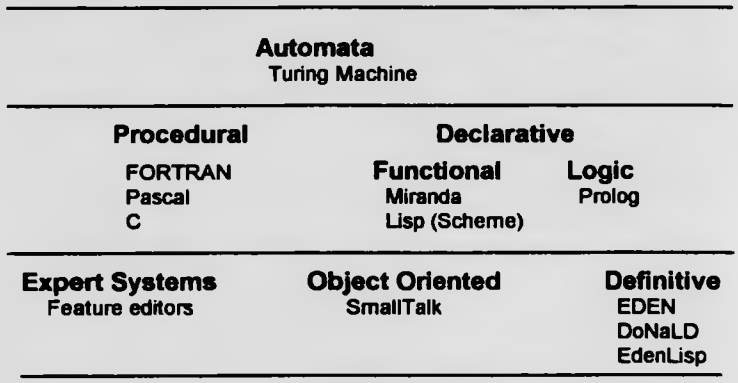


Fig 3.2 Some examples of Programming Languages
Levels denote increasing complexity

3.22 Functional Programming

Functional Programming languages (such as *Miranda* [Turner, 1987] and subsets of *Lisp* such as *Scheme* [Abelson & Sussman, 1985]) provide tools that emphasise the 'stateless' nature of computing. In those languages there are no changes because there is no concept of variables changing values.

> 'The question of whether the use of a Name in an Expression means the Name itself or the value to which the Name refers is meaningful in a language that has named memory cells (*i.e.* variables) because the value in a memory cell might change. In a [functional] language with no memory cells the result is the same whether we define new Names to replace Expressions or replace Names by the Expressions to which they refer. This is known as "referential transparency".' [Glaser, Hankin & Till, 1984]

The virtue of functional languages is the lack of explicit sequence of control flow in the program, which relieves the user of the burden of specifying the control flow. All that is required in order to specify a functional programming (FP) system, according to [Backus, 1978], is to specify the following sets.

- Atoms (*e.g.* digits, characters)
- Objects, derived from Atoms (*e.g.* `<YES, NO, 123>`)
- Primitive Functions over Objects (*e.g.* `null, id, eq, >, <,transpose`)
- Combining Forms (*e.g.* `composition, choice, insertion`)
- Definable Functions derived from the Combining Forms,

These sets should suffice to specify a problem completely, but unless there is some way of remembering Definitions the problem fixes the choice of primitive functions and Combining Forms. So the FP domain is extended to include Applications, in order that a function can be applied to an atom explicitly. That then raises a further difficulty: how do we remember Applications? A facility is required for allowing one to fetch a function from a library by giving the name by which the library would know that function; one would also need to be able store such functions in the library. However in accessing the library an output may not be solely dependent upon its inputs (e.g. time of arrival of inputs may be delayed indefinitely). That addition of non-determinism appears to invalidate the property of referential transparency. It is the issue John Backus called "history sensitivity" and is a serious problem in functional programming, for it also suggests state in a stateless system. One suggestion, according to Glaser *et al*, [*op cit.*], is to introduce special non-deterministic operators to deal with the particular issue they call "fairness" whereby if several inputs potentially demand infinite resources none of the objects get held up waiting for resources: there is sharing of some kind. The problem of state remains an area of research in pure functional programming.

3.23 Logic Programming

An alternative approach in the same declarative mould as functional programming (FP) is Logic or Relational Programming (RP), of which Prolog is the most popular instance. The fundamental difference is that whereas FP functions are many-to-one, RP specifies many-to-many transforms. In RP there is a set of solutions to a particular application rather than a single solution produced from a function application. So the statements

```
Joe is the brother of Jack
Jack is the brother of Fred
Joe is the brother of Fred
```

allows Joe to be mapped to two different siblings; that would not be allowed in FP.

Logic programming treats relations without regard to direction - *i.e.* which is computed from which. Programming is driven by queries about relations. The user supplies the facts and rules, the language used deduction to compute answers to queries. Thus we can write

algorithm = logic + control

'Logic' refers to the facts and rules for the algorithms, 'control' to how the algorithm can be implemented by applying the rules in a particular order. The user provides the logic, the language the control. Control is characterised by two

decisions: goal order - choose the left most subgoal - and rule order - select the first applicable rule.

Both FP and RP are non-procedural and involve programming without side effects. (A side effect is programmed if some element of the system gets altered in the course of running a function but is not itself the returned value of the function; e.g. a global variable gets altered or a screen change is made). In practice it is difficult to make a pure declarative language of practical use with-out some side effect. The very acts of interaction with a screen, printer or disc are really side effects. The 'value' of `print x` is undefined, although its side effect is not! In [Clocksin & Mellish, 1987] the issue is mentioned in passing in the discussion of "built-in predicates". Those facilities have side effects. "Satisfying a goal involving it [a built-in predicate] may cause changes apart from the instantiation of the arguments. This of course [*sic*] cannot happen with a predicate defined in pure Prolog". One fundamental built-in predicate that runs through the whole of Prolog logic is the *cut*, a pragmatic programming action to break infinite loops and chop useless searches in logic. The programmer's action inserting a cut is interesting - is it a state-based action?

The point seems to be that 'state' links computational logic and functional programming to the real world. It implies that computation cannot usefully exist without some reference to state. That accords with our earlier philosophical discussion that objects in the physical world 'exist' not just abstractly but also according to perception, and that perception is influenced by its history.

3.24 Procedural Programming

We introduce this section by endorsing prefatory remarks in [Abelson & Sussman, 1985]

> 'The computer revolution is a revolution in the way we think and in the way we express what we think. The essence of this change is the emergence of what might best be called *procedural epistemology* - the study of the structure of knowledge from the imperative point of view, as opposed to the more declarative point of view taken by classical mathematical subjects. Mathematics provides a framework for dealing precisely with notions of "what is". Computation provides a framework for dealing with notions of "how to". '

Procedural programming can be thought of as writing a recipe for solving a problem. Where it differs from pure functional programming is that variables can be assigned values and those values get changed during the program. From that

point of view the program can be said to have state. However the nature of that state is not so intimately bound up with the state of the problem being programmed that any instantaneous state of the computation tells something about the state of the model.

Consider a common procedural programming language such as Pascal or C: a procedure is written as an algorithm that produces required outputs from a given input set. If the procedure is halted during execution and internal variables examined, their instantaneous values do not necessarily have any relation with the final result. For example, suppose there is a loop programmed in Pascal in the form

for *i* :=1 **to** 100 **do begin** ... **end;**

At a point of interruption the value of integer *i* is of interest purely to see how many times the algorithm has looped. The internal 'state' of *i* in the procedure is irrelevant as a state of the output. (Similar observations can be made about data storage references such as addressing array elements.) Procedural methods at this level simply provide a recipe for describing a resulting state that then has 'meaning'.

A further difficulty comes with any slight change in the real system being modelled. Often a new abstraction is required. Detailed examination of a problem frequently means rewriting the system, something disapproved of by commercial vendors of software. What is generally done is to have a static (stateless) system that is capable of describing states via data, and standard or preconceived ways of manipulating that data. In relation to design we shall see in chapter 5 that many established design support systems are programmed employing computer methods of that type. That means that "experiment" is not really part of the programmed system. The programmer of the system may be able set up environments in which experiment may be carried out, but those experiments have to some extent to be anticipated in order for the environment to allow them. For the shortcomings of that approach one merely needs to glance at the huge market in "enhancements" and "customisations" that are parasitic on most popular software.

3.3 State in higher level Programming

3.31 Data Modelling and Rule Based Systems

One approach to the difficulty of anticipating user requirements is to increase the size of the 'problem universe' by intelligent investigation of user requirements. The

question then is what limits apply to that increase. There are two independent constraints that can be applied to obtain computable sets from descriptions of objects and their infinite number of possible states, namely *time to compute* and *space* (memory or storage). Even if certain algorithmic descriptions are theoretically computable, in practice many turn out to require either time or space which for practical purposes is too large to be feasible. If we place restrictions we can define an important class of representations that are both time-feasible and space feasible, and solutions to problems in those categories can be implemented by trading off storage against time. A useful restriction is to specify a domain by knowledge type. We can group and store knowledge according to methods that humans use, *i.e.* by rules of thumb: we must then store the methods of structuring the domain too. Programs of that type are called expert systems. The rules of thumb are known as ***heuristics***: rules for structuring based upon human percepts, such as those suggested by the structure of Minsky's Frames.

[Michie, 1986] examines the knowledge content of expert programs. He shows that in a space and time limited computer system the amount of knowledge depends upon what other things must be stored and/or use computational time. At its simplest, knowledge may be thought of as a finite function $f:X \rightarrow Y$, i.e. of a look-up table of pairs of the form $(x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots, (x_n, y_n)$ where N is the size of the domain of f. Each pair represents the smallest possible transition that adds to the knowledge store. If we structure the transitions by heuristic rules, patterns, descriptions, etc. then those structures Michie calls advice. His figure, reproduced as *fig. 3.3*, shows how the increase of advice and the need for a control program effectively reduces the knowledge content in a fixed store. This is the fundamental problem for expert systems.

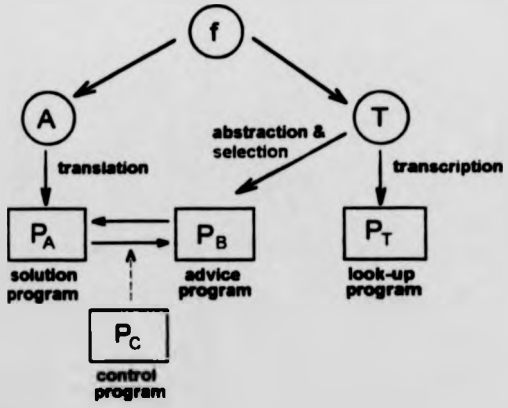


Fig 3.3 Relationships among various kinds of mathematical and computational objects. Abstract objects are ringed, concrete are boxed. A and T denote two contrasted abstract representations of the function f, namely as an evaluation algorithm and as a function table (ordered set of pairs), respectively.

[Reproduced from Michie, 1986, Ch. 18]

In [Koegal, 1989] this point is endorsed: 'Although expert systems have been developed to perform design in specific domains, such systems are difficult to adapt to other domains.'. The way round that suggested by Koegal is to make the advice section structure the domain in hierarchical chunks: abstracting at a higher level, *e.g.* to separate 'planning' from 'planning this'. Thus a design gets developed in stages of decreasing abstraction.

'Advice' in the context of object or real world description consists of heuristics structuring object and state descriptions, but not actually creating objects with state. A system must therefore be 'content' limited, using content in the form described earlier. Indeed Koegal's major point is that expert systems are at their best when the form of the design is already in the designer's mind *[op cit.]*. If *form* can be delineated then the task becomes that of abstraction on a data or knowledge base. Initially much work was done implementing specific domains for helping design, mainly using declarative programming, which is a form well suited to formal rule-based systems. Implementation meant providing a suitable vehicle for structuring large data bases that form the backbone of editors for aiding design. [Dym, 1987] provides a useful introduction to the problems of engineering applications of expert systems. Shape grammars [*e.g.* Leyton, 1988] have helped in the development of shape editors that are now big business, usually called Feature Editors, implemented to reduce input time in repetitive modelling activities. Examples of feature editors are many. [Gu, El Maraghy & Hamid, 1989] give a useful introduction with a good reference list as well as an example of the genre.

Using form features many common geometrical shapes may be generated and placed in libraries together with appropriate structuring heuristics, rather like a thesaurus in word processing. Whilst it is easy to decry the use of shape grammars and feature editors as "trying to write an essay using only a thesaurus and a dictionary" there is a large set of routine design tasks that, like entering data base material, is fairly standard. However as a way of modelling objects with state the approach is clumsy and requires sophisticated tools for adding to the 'advice' box. And advice that is useless is worse than useless as it displaces useful knowledge in a finite system.

In more recent work using rule-based methods modelling the modelling task itself forms the basis of design support tools. [Krause, Vosgerau, & Yara-manoglu, 1989]. Hierarchical dissociation is a feature of most design tasks so product data models with suitable abstractions of structure will enable the delineation of the

form of the design to be systematised. The Leeds Structure Editor is an example of that approach, where a generalised software tool is used to create product data models with semantic data model characteristics [McKay, 1988, Shaw, Bloor & de Pennington, 1989]. Examples are quoted there of characteristics that are fundamental to semantic data models: unstructured objects, relationships, abstractions (under which is listed classification, generalisation, aggregation and association), networks of hierarchies, derivation/inheritance, editing of constraints and dynamic modelling. The Leeds work, along with similar rule based systems deal with static data models. Dynamics and the problems of state transition are problems still.

3.32 Object Oriented Programming

One way to characterise an object state is by having one or more *state variables* to maintain enough information to determine the current behaviour of the object. If those variables can be *encapsulated* into a procedure then we have procedures that have history, procedures with state: a computational object that has its own state variables that can change over time, *i.e.* as the program runs. To do that in a declarative language we need to introduce an assignment operator that enables us to change the value associated with a name. (For example in Lisp we have the assignment operator *setq* that a 'pure' functional version of Lisp would not have.) Associating the assignment operator with local variables in a procedure allows us to create computational objects with local state. Effectively each time a procedure is run it changes its state. For example a procedure modelling a bank balance simply subtracts an amount withdrawn from the current value in the account. The procedure alters its own internal variables, *i.e.* it is self modifying. Those variables are not generally accessible from outside the procedure. That is the principle of encapsulation. Methods for accessing the variables have to be specially written into the procedure.

Computational objects form the basis for the higher level languages called *Object Oriented Programming* (OOP). In this style of programming, objects are created that have local state because they have *encapsulated* or hidden variables. Neither those variables nor their values are directly accessible by other objects or procedures. At its simplest such an object may consist of a single procedure but they usually have associated other properties called "methods" described below. Computational objects may be used to model rather than simply describe physical objects. An abstract object group such as 'circle' may be defined generically in terms of user-defined general words, and general properties of the 'class' of objects

specified by the user. At a second level instances of objects may be given attributes that are more particular (e.g. circleA of radius 4 mm). A class of object is associated with a specific set of operations called 'methods'. Methods are procedures that have well-defined inputs and outputs. The way that the Methods are written is irrelevant - indeed the user is not intended to know how they are written. Each item of data within a program is regarded as an attribute of some object and only accessed by invoking one of the methods defined for the class of that object.

OOP languages have been developing since the introduction of Simula 67. The seminal OOP language is Smalltalk-80, [Goldberg & Robson, 1983]. Since that time most popular computer languages have developed the form, including C++, ADA, Modula-2, and the declarative language Lisp.

Information hiding (encapsulation) is a crucial aspect of OOP. If a Method has to compute the value of a useful physical variable within its algorithm in order to complete its output, that intermediately computed variable is inaccessible to the user. (For example a method for a generic beam-in-bending class may involve calculating a section modulus; but if it is not a 'message' that section modulus cannot be obtained from the 'method'). However, if information hiding is essential to OOP, it also proves to be a limitation. Tomiyama identifies the problem of extending its application where there are objects that share a common connection.

> 'Let us consider the design of a robot. In Smalltalk-80 it is reasonably natural to say;
>
> - a_Robot is an instance of the class Robot
> - a_Robot has 6 arms, arm1 to arm6, as instance variables.
> - arm1 is an instance of the class Arm
> - arm has instance variables, end_Point1, end_Point2, length, transformation Matrix, *etc.*
> - end_Point1 is a_Point which has x-, y- and z-coordinate as instance variables.
>
> Problem
>
> How do we define the fact that end_Point1 of arm2 is the same as end_Point2 of arm1? Smalltalk-80 does not allow sharing instance variables among two objects because of information hiding.' [Tomiyama, 1989]

Despite the problems described here some very profitable work on design environment has been done and those will be discussed in chapter 5. Basically the methods create artificial environments by storage of large amounts of information and relationships on that data that are based upon observation by the users

3.4 A New Programming Paradigm for State

3.41 Background to Definitive Notations

The issues that have been discussed in this chapter relate to the computational context of this thesis. It has been shown that while state and state change are crucial to design, they are difficult to have using conventional programming methods. Definitive methods on the other hand have distinct properties of state that provide great promise in application to design.

Early work on definitive notations was orientated towards "the state of the interaction". This is in contrast to traditional procedural and declarative programming methods that were developed originally for interacting with the computer in a batch mode. In the latter methods, acts of input and output were simply regarded as necessary evils to get into the stateless mode described above.

To exemplify these properties two early definitive notations, called ARCA and DoNaLD, are reviewed. The first is named after Arthur Cayley and is for animating Cayley Diagrams. [Beynon, 1986]. Graphs and similar constructs such as circuit diagrams and control system block diagrams carry a content that is a function of the incidences on a diagram, but can only be inspected after the diagram is constructed. The visual image has little value without the underlying conceptual model, so it is necessary for the user to be able to specify the models underlying the images systematically and simply. In such a task interactions are highly important and need to be mediated through both graphical and textual interfaces. The text screen (or window) is used to develop the program code, whilst the graphical display shows the current state of the model. The code consists of a sequence of definitions. A definitive notation includes variables that denote implicitly or explicitly defined values in the underlying algebra. Values of variables are determined by definitions, each of which either assigns a formula or a specific value to a variable. Circular or recursive definitions are trapped as semantic errors.

To the user a definitive notation appears similar to a spread-sheet, without the cell-based interface normal to spread-sheets. Actions are essentially a dialogue with the user consisting of declaration of variables (if variables are typed), definition or redefinition of a variable, and evaluation of a variable. Writing a definition is like putting a value or formula in a cell of a spreadsheet. Redefinition simply over-writes the old definition in that cell. Another important similarity is the way that computation proceeds to update the system after a definition is input. Computation that takes place with a definitive notation has the following properties.

a) It is hidden from the user,

b) It occurs after every definition and updates values of variables in every previous definition affected by the latest (current) definition and

c) It can be carried out in any order whilst updating previous definitions.

An interesting feature of definitive methods is that it is not 'pure' (purely procedural or declarative). In ARCA, declarative features (in the form of constraints) and procedural forms (means of updating coordinate and incidence information) are mixed. This is a characteristic which the method shares with a number of programming forms such as Sketchpad and PopLog. The advantages of that are great and have been increasingly exploited as the method has been developed.

DoNaLD is a geometrical implementation of definitive methods: a 2D line drawing notation [Beynon, Angier, Bissell & Hunt, 1986]. It is a typed notation. The underlying algebra is based upon **real, integer, point, line** and **shape** variables, where **point** values are pairs representing Cartesian or Polar forms and **line** values are line segments in a plane. A **shape** value is a line drawing consisting of a set of lines and points. Objects designed in DoNaLD consist of shapes defined by a set of definitions. Its power lies in the ability to construct abstract shapes since values of variables can be specified by algebraic expressions. So if the user inputs particular values as input to those expressions the realisation of the shape reflects those assignments. An example of DoNaLD script [*ibid*] follows.

```
openshape cabinet
within cabinet (
      int    width, length              # declaration of variables
      point  NW, NE, SW, SE
      line   N, S, E, W

      N = [NW,NE]                       # abstract definitions
      S = [SW,SE]
      E = [NE,SE]
      W = [NW,SW]

      width, length = 300, 300          # value assignment

      SW = (100,100)                    # abstract definitions
      SE = SW + (width,0)
      NW = SW + (0,length)
      NE = NW + (width,0)

)
```

In the example a simple square is defined in terms of points for corners and line segments for edges. Data types are interpreted as indicated. Variables defined within **openshape** are deemed to be local. We thus have an object with local state defined in terms of the values of length and width. Since both are defined in the example the square computes and the object appears on the graphical interface. If either `width` or `length` are redefined then the square recomputes and a new object replaces the old. Notice that although a square is actually drawn it has a general object label `cabinet`. Within `cabinet` other attributes may be added, so `cabinet` gives the underlying definition of the object - the graphics are simply the representation.

3.42 State in Definitive Notations

How then can the definitive method be applied to design? A primary issue has to do with state. A family (or *script*) of definitions shows that the state of interaction is linked with the concepts of *object state* and *process state* discussed above. Indeed we have a method that captures computational state. As such it can be a vehicle for representing physical state. A script of definitions establishes relationships independently of the data required to define a particular state fully (provided there are no duplicate or cyclic definitions). Thus the script has *abstract* state rather like OOP objects. Unlike OOP, state variables are not encapsulated, rather they are a visible part of the user's interaction. Changing variables (modelling state transition) is not done by message passing but directly by redefinition.

Redefinition is like trying something out in design - trying an alternative approach to an already tried scheme. It is something that is typical of user interaction. However the designer wants more than to be the sole source of redefinition. When a particular action is taken the user may want to be able to see the result in different ways. We have already seen above the role of side-effect in displaying on screen the state of the dialogue. The representation shown on the screen may not be necessary for the computational model. It may only be like a photograph, a snapshot of particular aspects of the model. If the model changes then the role of the computer in updating the representation on screen is in a sense independent of the user. The screen display should not only be able to reflect the new state but also to present that new state in the most advantageous manner. Thus traditional user-interface management issues come to mind: the computer should be the source of dialogue though appropriate use of windows, menus, graphical displays and the use of analogue rather than textual input, [Foley]. Of equal importance are

issues such as monitoring and maintenance of constraints, an activity in which the computer must itself participate in changing the state of the dialogue. Those desires imply that the computer should not have the passive role of simply responding to user redefinition, it should have a more general framework.

In understanding how the computer can have a more active role in redefinition, the idea of scripts being computational objects is a useful device. A definitive script may be thought of as an object that gets changed either by the user or by redefinition by the computer. It would therefore be best if the same method can be used regardless of the source of the redefinitions. The method proposed is based upon an Abstract Definitive Machine model (ADM) described in *Fig.3.4*. The model consists of an overall *Program store P* containing *Entities* similar to objects in OOP. Each entity has two stores: *Store D* of *variable definitions*, and *Store A*, which contains *Actions*.

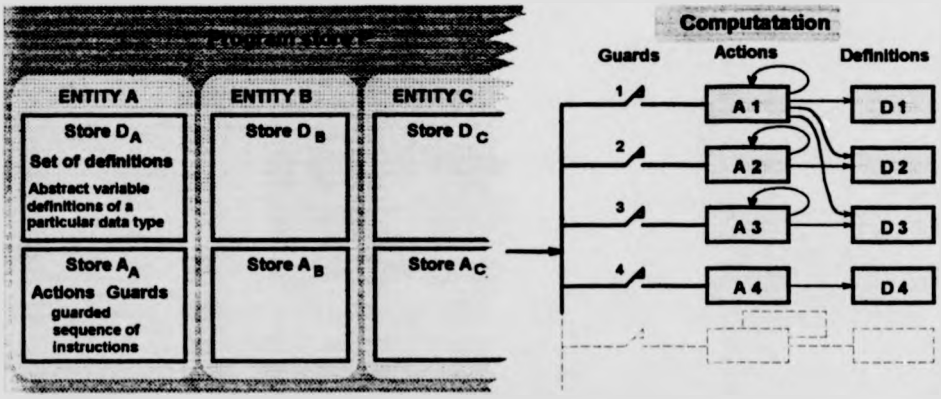


Fig 3.4 Abstract Definitive Machine Model for Object Definition

The definitions in store D are sets of definitions similar to those in the DoNaLD example - namely variable declarations of appropriate types and definitions consisting mainly of abstract formulae, but may also include assignments of values to variables. The new addition to the definitive notation is the idea of an *action*. An action is a sequence of instructions that may do one or more of the following

- redefine a variable
- introduce a block of new definitions sitting somewhere else outside the current entity
- delete a block of one or more definitions

Actions affect the contents of Store D and Store A. The triggering of actions has to be carefully guarded to prevent infinite loops and actions interfering with one

another. These guards may be regarded as Boolean switches, shown as such in the right hand of *fig.3.4*. A computation consists of a sequence of parallel executions of appropriate actions. All the guards on the actions are evaluated and those that evaluate to *true* allow the actions behind them to occur. The effect of an action described above is to modify the contents of either or both of stores D and A within an entity. Actions may also affect other stores (as indicated by the dotted output of A1 in the right hand diagram).

The Abstract Definitive Machine (ADM) model provides a framework that retains the characteristic features of definitive programming, *viz*. the representation of the current state of the user-computer interaction by means of a system of variable definitions, but allows both the user and the computer to initiate dialogue actions to change the state.

These important issues in the design and implementation of CAD systems are discussed further in [Beynon & Cartwright, 1989].

The abstract machine model is not itself a programming language or notation: it is just an idea for structuring a definitive programming system. It does have features relating to the state of objects that are similar to many described in this chapter. However some clarification of the abstract machine is required as a step towards the longer term objective of developing a CAD support system. In particular the way that actions operate is not at all clear, and the problems of constraint maintenance may prove to be damaging in an engineering context. After all if one has many agents acting in a system constraints could easily be circumvented, modified or be just too complex to be implemented in a reasonable time frame.

In the next chapter we review some programming methods and design support systems used in the engineering scene that are definition-based before embarking on the application of definitive methods to geometrical modelling and interactive design.

4

Computation as Experiment

In this chapter definition based systems for geometrical modelling are reviewed before the work on applying the Definitive principle to design is described. The historical development of definition based systems reflects the concern of users to have geometrical models to help to shape design ideas. The emphasis of most systems seems to be more on actually displaying geometry than the shaping of ideas. Even recent systems that encourage experimentation do not exploit the rich potential of definitive methods. It is shown that the process of describing the state of a geometrical object enforced by the Definitive method is more in accord with the conceptual design process. The abstractions of shape required by the definitive method tend to have great flexibility and a potential that extends far beyond the solution of the particular problem for which they were built.

4.1 Definition-based Geometrical Modelling

Geometrical modelling began very early in the history of computing. Geometrical models that are used for design have a 'content' well outside their visual impact. In that respect they resemble graphs, circuit diagrams and so on. The designer often uses the displayed image as a guide to the state of the underlying design and the stage reached in the interaction. So it is important to be able to represent geometry. Curiously, one major early form of geometrical modeller based on definitions, APT, was not aimed at design but at manufacture, where geometry is already well defined. The reason was probably because it was text based, antedating computer graphics. APT (Automatic Programming of the Tool) was developed at MIT in 1956 for producing machine tool control instructions for the

shaping of aircraft structural sections: wings, fuselage, *etc.* It was structured around algebraic definitions of shapes of surfaces. Graphics based geometrical modelling came in as Computer-Aided Design systems were developed. Initially Computer-aided *Design* was a misnomer as systems were oriented towards draughting. Such draughting systems remain popular, still primarily oriented to 2D, although 3D is gaining ground. Geometrical modellers in 3D are based either on constructive solid geometry (CSG) or boundary representation (B-Rep). The core of many commercial systems owes much to another definition based system PADL-2.

The user interface of most CAD systems is command-based and can be programmed to create standardised command sets often called parametrics. The latter has recently become important to vendors with the growing realisation that customisation is vital to most applications to make real savings in productivity. We need to examine these developments with the definitive method in mind.

4.11 APT

APT began as a library of FORTRAN routines to solve algebraic equations. Its function was to define an object in terms of surface shape characteristics in such a way that a set of 3D coordinates (in any coordinate form) could be output. Those coordinates would be input to a machine tool control system that would cause a tool to move from one programmed point to the next so as to produce the surfaces. Tool movement could only be controlled initially by point to point in a straight line. Later, interpolators were introduced; linear, circular and exponential interpolation allowed continuous path profiles very close to that desired. Accuracy of form is achieved by setting a suitable interpolation step size in the coordinate output

The definition basis of APT is illustrated in the following program example from [Koren, 1983]

The Part program for the geometry is as follows

```
10 SETPT = POINT/0,30,25                    $ Start position of tool
20 CNTR = POINT/110,90                      $ Centre point (100,90)
30 CRCL = CIRCLE/CENTER,CNTR,RADIUS,30      $ Circle at CNTR radius 30
40 LFTSID = LINE/(POINT/80,40),LEFT,TANTO,CRCL
                                            $ Line (80,40) to left tangent of CRCL
50 PTB = POINT/140,40                       $ Point (140,40)
60 BASLIN = LINE/(POINT/80,40),PTB          $ line from (80,40) to PTB
70 PTM = POINT/140,90                       $ Point (140,90)
```

```
80 RITSID = LINE/PTB,PTM                          $ Line from PTB to PTM
90 TOPLIN = LINE/PTM,(POINT/110,120)              $ Line from PTM to (110,120)
100 AUXLIN = LINE/(POINT/140,120),RIGHT,TANTO,CRCL
                                                  $ Line (140,120) to right tangent of CRCL
110 XYPLN = PLANE/CNTR,PTB,PTM                    $ Plane through 3 points
120 PSURF = PLANE/PARLEL,XYPLN,ZSMALL,15          $ Plane parallel to XYPLN 15 mm below
```

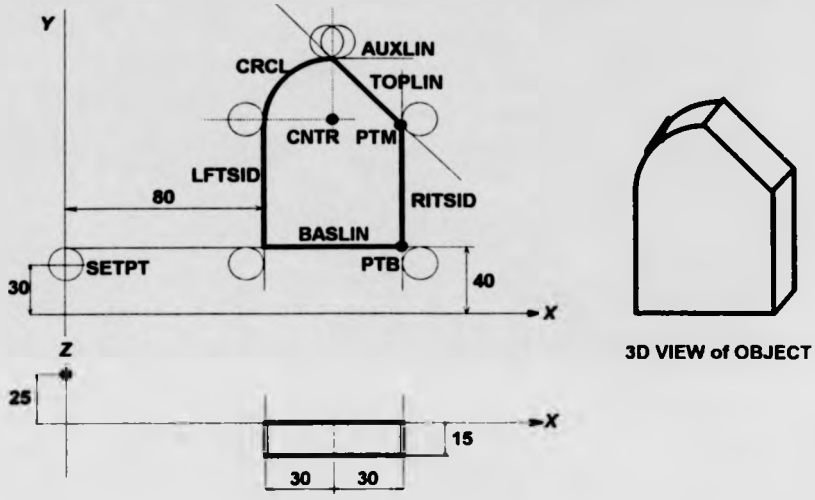


Fig. 4.1 Geometry of Program Component [from Koren 1983]

Language words denote geometry, *e.g.* POINT, LINE, CIRCLE, RADIUS, RIGHT, TANTO. These words may carry modifiers, following a slash, to define particular values. Modifiers may be expressions with references to other variables or definitions. Definitions are not typed: users are expected to check that by inspection.

User names are in FORTRAN style: upper case alphanumerics of up to six characters. Names used in the program are shown in *fig. 4.1*. The part program is submitted to the APT processor and the output is a list of coordinates marking end points or turning points. In the case of circular arcs the output lists the end point as *(x,y,z)* and the centre offset from the start point of the arc as *(i,j,k)*.

By itself the coordinate list is incomplete for machine tool purposes. Geometrical statements need to be turned into machine motions. That is done by the addition of 'technological' commands to the Part Program. For example the above program continues as follows.

```
210 CUTTER/20.0                   $ Cutter Diameter
220 TOLER/.005                    $ Tolerance 0.005 mm
230 SPINDL/1740,CLW               $ Spindle Speed 1740 rev/min clockwise
```

```
240 FEDRAT/2500                          $ Rapid Feedrate 2500 mm/min
250 FROM/SETPT                           $ Move From Setpoint
260 GO/TO,BASLIN,TO,PSURF,TO,LFTSID      $ Go to BASLIN drop to part surface at LFTSID
270 FEDRAT/500                           $ Change feed to 500 mm/min
280 GO FWD/BASLIN,PAST,RITSID            $ Commence cutting along BASLIN to RITSID
290 GO LFT/RITSID,PAST,TOPLIN            $ Machine up RITSID
    . . . . .                            $ Program continues ...
```

The addition of these statements gives an output from the APT processor that includes machining codes. Those codes are still unusable since each machine tool is different. The codes must be input to a post-processor appropriate to the local machine tool settings.

APT has historical importance; in modern CAD/CAM systems there is no need for the geometry to be defined this way as the coordinate information is already in the CAD data. For our purposes it is interesting to note the definitional and command forms that the input takes. Computation is of course on a static model - changes in geometry have to be recomputed.

4.12 PADL-2

The shape of the part program of APT could well have influenced one of the earliest forms of solid modellers: PADL. Its first experimental system it goes back to 1975. It handles objects describable as combinations of orthogonally positioned blocks and cylinders. It is important in the present work as its form at first sight is quite close to the definitive method, although differing fundamentally in significant aspects.

PADL-2 was introduced as an industrially viable "core implementation" of CSG, funded by the American National Science Foundation and ten industrial sponsors including Boeing, DEC, Eastman Kodak, McDonald Douglas and Tektronics. The core has been put to diverse use apart from straight solid modelling systems: CNC machining simulation, [*c.f.* Tan, *et al*, 1987], verification and programming; simulation of industrial robots, representation of dimensions and tolerances, automatic feature extraction; machine process planning and automatic adaptive finite element mesh generation and analysis. Several commercial systems incorporate PADL-2, including Unisolids (McDonald Douglas), Cynergy (Westinghouse), Series 7000 (Autotrol) and AutoSolids (AutoDesk). The user manual gives the following introduction [Hartquist & Marisa, 1983].

PADL is an acronym for Part & Assembly Description Language. More generally, PADL has come to designate a family of languages and geometric (solid) modelling systems developed by the Production Automation Project at the University of Rochester. The primary representational medium in all is CSG.

The PADL language is text or keyboard oriented media with two types of statement

• declarative (definitional) statements, e.g. for creating and editing geometry, as illustrated by all but the last of the statements in the program below, and

• imperative (command) statements for evoking actions, e.g. generating displays via the DISP commands.

The following is a sample program in PADL-2. Line numbers are not part of PADL-2.

```
1.    GENERIC EXAMPLE (FINAL)
2.    A = 5
3.    B = 10
4.    C = A + B
5.    D = CYL(H=A,R=C)
6.    E = BLO(X=1,Y=B,Z=C)
7.    CS1 = MOVX=2,MOVZ=3,MOVY=10
8.    CS2 = CD1,MOVX=1
9.    OBJECT1 = E MOVEDBY CS1
10.   OBJECT2 = D MOVEDBY CS2
11.   FINAL = OBJECT1 UN OBJECT2
12.   DISP PART
```

Commands tell PADL-2 to do something immediately. For example, DISP causes the interpreter to run through all the definitional statements and do the calculations, and then to display the current set of defined objects; SHOW lists all the statements input into PADL in the current session; other commands such as SET ACCURACY = 7 will change the internal settings of PADL-2. None of these commands is remembered at the conclusion of a session.

Definitional statements are the basis of the PADL-2 language. They define user parameters (e.g. names of objects) and then assign them to the different primitives, coordinate systems and movements. The form of definitional statements is

```
username = <definitional expression>
```

and assigns a meaning to the username. Username can be any alphanumerics not commencing with a digit and not otherwise defined as linguistic terminals. Letters are always translated to upper case internally. There are three types of definitional expression corresponding to the three data types it supports:

- real expressions (may also be Boolean expressions) (e.g. lines 2, 3 and 4)
- solid expressions (e.g. lines 5 and 6)

- parameter lists (may be coordinate systems or motions)

Statements are stored within PADL-2 and are only calculated if some sort of display command is issued. Their position in the statement listing, or whether they are needed to display the particular object asked for, does not affect whether they are calculated or not. Redefinition of any of the parameters can be done by just typing in the new value: entering `A=6, B=12` discards the previous values of `A` and `B`. Any future displays or calculation on the object is done with the new values unless names are self-referential or cyclic in which case an error is indicated. A definition sequence defines "static" entities, not the "dynamic" entities that one can create with programming languages. A sequence is therefore represented as an acyclic directed graph. [Hartquist *et al.*, *op cit.*]. PADL-2 statements are not saved in the processor as entered. Only definitional statements are stored: they are translated into graph format and reconstructed if requested (*e.g.* by the `SHOW` command) but without redundant parentheses, white spaces or comments. Commands are executed as soon as entered: they are not stored.

Primitives are the usual CSG ones such as CYLinder and BLOck in the example. Derived primitives, called meta-primitives, may be constructed by employing infinite planes, half planes and existing primitives. Modifiers are set UNion, INTersection and DIFference of CSG objects. Coordinate systems are Cartesian with translations `MOVX`, `MOVY`, `MOVZ` along, and rotations `ROTX`, *etc.* about the coordinate directions.

An object in PADL-2 is formed by a set of definitional statements. That set may be identified by using the command `GENERIC <FILENAME> (<FINAL OBJECT NAME>)` at the head of the set of statements. In that case the command `DISP` without the generic name will assume the object name in the latest generic. Superficially a `GENERIC` resembles the definitive part of the 'entity' described in section 3.4 above. 'Actions' in the definitive context are effectively alternative definitions that depend upon current values of other variables. In PADL-2 one could perhaps model a simple 'action' by means of the conditional expression. For example the statements

```
Z = 10
A = IF Z GT 10 THEN 1 ELSE IF Z=10 THEN 2 ELSE IF Z LT 10 THEN 3
```

set `Z` to the value `10` and then `A` to `2`. If subsequently `Z` is redefined to `15`, say, then `A` resets to `1` after the next recalculation. A definition sequence may be reset by the guards in the conditions. That allows some variations in generic objects. The

system is still essentially static as the evaluation of A depends upon the current value of Z and is only evaluated when a display is requested. Thus the `if` statement is not really an action since the definition itself remains unaltered after the action

In a recent examination of PADL-2 by Helen Butchard [Butchard & Cartwright, 1993] she found most of the difficulties were associated with the inability of the system to store commands executed during a session. Each time a geometrical object is to be produced, or PADL-2 is initiated, all the set-up commands have to be re-input; *e.g.* for the display: colours, accuracy and view type; and for output formatting: type of output file, *e.g.* Postscript. Whilst commands could be included in an input file they were only actioned at the time they were actually input. Subsequent calls on commands, to display for example, had to be re-input when required; the command SHOW does not list any commands.

NOTE Other practical problems occurred using PADL-2. The definition of coordinate systems quickly becomes confusing. It proved better to define systems in terms of one another rather than absolutely. Computational time is a serious problem in PADL-2: Display time is great and Boolean operations seem to take a great deal of time to compute. [Brown, 1982] gives a more thorough technical review of PADL-2.

4.13 Parametrics

Commercial CAD systems generally have an interface language that allows users and developers to write macro programs and functions in a high level form that accesses the command structure indigenous to the use of the CAD system itself. These languages are well formed in that they correspond closely to popular programming notations such as Lisp, Forth, Basic and C. The most common way of using these languages is to create sets of functions to do tasks that customise the basic CAD system to local needs. Straight customisation might be to adapt the menu structure or make specialised functions available on menu. The use of these languages to create abstract objects is called parametric modelling. The user creates an object on the CAD system and then proceeds to assign variable names to particular dimensions or shape features. That enables a 'parametric' to be written to produce the drawing with different dimensions. As the given program is run it requests values for the parameters required to construct the drawing.

The form of interface languages is generally complex and requires a good knowledge of the CAD system itself. Because the languages sit 'on top' of the

CAD system they are computationally expensive and inefficient. Nevertheless they have proved a very strong spur to progress in object representation and a substantial commercial industry has grown up alongside all popular CAD systems.

Parametrics serve the need implicit in PADL: being able to specify objects in ways that are flexible in their design features. In concept they also share the limitation of being static objects inasmuch as they describe an object rather than create an object with state. However more recent parametrics, on ComputerVision for example, allow animation and state changes that are more akin to the definitive approach. These owe their development to work on standalone systems of which Design View is typical.

4.14 DesignView

The limitation of not being able to use parameter changes to change the object representation automatically is clearly surmountable. DesignView® [DesignView Manual, 1991] is such a solution. A quotation from the manual gives the flavour.

> DesignView is based on dimension-driven variational geometry, a technology that makes drawing much easier and more flexible than ever before. When creating a drawing using DesignView, you do not need to be concerned with the initial sizes of geometric objects - you need only sketch in the basic shape. Later on, you specify the exact dimensions and DesignView automatically reshapes the geometry for you. DesignView's dimension-driven variational geometry maintains complex relationships between drawing elements, so circles stay tangent and lines stay connected.
>
> DesignView's powerful analytic capabilities can solve inequalities and simultaneous non-linear equations, calculate mass properties, simulate dynamic systems, and more. It is ideal for synthesising linkages, doing tolerance stack-ups analysing forces and solving complex geometric problems such as belt-pulley configurations.

DesignView is written to suit a graphics system so can sit easily on top of an existing CAD system. The PC version sits on a variation of the Windows simple line drawing package, extended to allow greater graphics manipulation (*e.g.* the creation of splines). Another version sits on CADDS5 from ComputerVision. A sample picture from the manual illustrates the method and is quoted as *Fig. 4.2* overleaf.

The problem illustrated in the figure is to design a wall mounted crane using force polygons. The user draws in the shape of the crane with some particular

dimensions and also constructs the stress diagram. The system provides the graphics facilities to do the drawing. Labels are added to the geometrical features that are to be constrained. Relationships in the form of equations on those labels are then written using DesignView facilities. The constraint system ensures that the geometry then reflects the current values of the variables in the equations. In the case of the crane the variations are the dimensions associated with the geometry and the load it must support. If, for example, the vertical dimension of element 5 on the crane is increased to 50 then the crane diagram is redrawn and the stress diagram is automatically updated to reflect the new loading regime. Similarly the results table is updated to note the new loading values. Successive changes may be input by means of iteration over a user-specified range. Those changes may then be animated to give the illusion of a mechanism in operation.

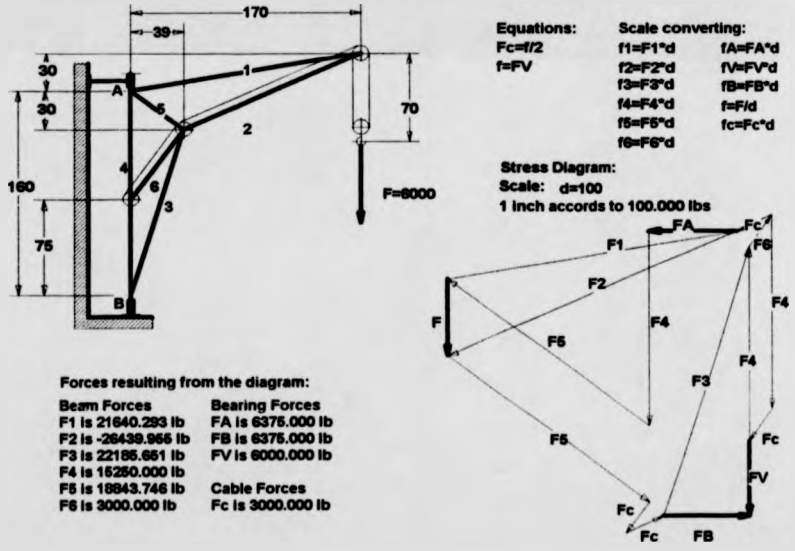


Fig. 4.2 Wall mounted Crane - file ex-crane.dv [DesignView Manual, 1991]

Because the program is constraint based it is not as flexible as it appears at first sight. Only pre-conceived parameters may be varied. One cannot for example change the appearance of the screen display through the system, nor can one change the dimensions to SI metric as neither of these was conceived when formulating the problem. Also the attributes that can be added to objects are merely recorded. Only relationships resulting in a geometrical change will be automatically updated. DesignView may be linked to external programs such a spreadsheet using a special connection that allows parameters to be manipulated in

the spreadsheet and then transferred into DesignView. In that respect the system is the nearest to the definitive method. It also points up the need for this type of facility

4.2 Definitive Notations for Geometrical Modelling

4.21 Comparison with other systems

Of the definitional methods we have described, DesignView is nearest to the definitive philosophy that we advocate, especially as it has an explicit link with spreadsheets. As an environment for design it has some desirable aspects. However it does bear the marks of a pragmatic approach to design of objects, essentially extending an idea incrementally. It is natural to try to strengthen existing commercial systems. Engineering applications are very demanding of computers - and ever more so as product life cycles are dropping so rapidly. The problem is that there is great inertia in the commercial systems because of the huge investment in programming. For example the decision by software developers to move from FORTRAN to C was made with great reluctance even though it yielded great gains in software development. Furthermore the success of approaches such as ProEngineer show that developers willing to try new methods can benefit. A new programming should be commercially attractive if it can be shown that it can be used to deal effectively with relating form and content in CAD and with problems of integration and interaction.

Definitive methods appear to hold out feasible solutions to those problems. The value of an interactive system to an engineer is not whether it can produce a solution to one current design problem: rather it is whether it can maintain selected relationships established in earlier attempts at a solution, whilst allowing the designer to try quite different approaches. That implies that the system must make all the current relationships available to the user, so that they can be modified and particular entities already designed can be referenced for future interaction.

Take PADL-2 for example. Referencing objects already designed may be done via previously defined names and composite objects by means of the GENERIC device. Modifications are relatively easy provided they use the primitives built into the system (or meta-primitives derived from those and bounding planes). As abstractions these primitives are simple, with default values (usually of unity) for all defining dimensions. The underlying algebra is of reals, solids and parameter lists. If we wish for example to reference vertices on an object that must be done

indirectly since they are unlabelled, although it is possible to 'cheat' by accessing the system's own internal record of the object. To be able to have interaction of any kind whatsoever, *e.g.* changing a solid representation to a B-rep or wire frame or a 2D drawing or changing the display to suit a particular representation, we need a programming method that is much more open.

It needs to be shown that definitive methods are applicable to those areas. We start with the representation of shape. My work demonstrates that definitive methods enable shape definition of great abstraction and flexibility, permitting the design of computational objects with potential that extends far beyond the solution of the particular problem for which they were built.

4.22 Representation of Shape

In order to be as abstract as possible in thinking about shape, geometrical models are first defined in terms of reference and construction points, labels, skeletal structure and geometrical operations that are typically used to synthesise complex objects from simple components. By that means use may be made of many different computer representations: whether it be wireframe, CSG, B-Rep or any other.

Wireframe constructs are easiest to model in abstract terms. The structure can be represented by labels, connectivities, linetypes and coordinate information. They are powerful models inasmuch as they can suggest a content far beyond the form represented, as the example in *fig. 4.3* shows. The letter E emphasised in three dimensions can be perceived in the form shown even though there is no connectivity to show a solid shape. Similarly "solids" may be indicated simply by shading or selective line removal.

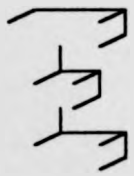


Fig. 4.3 Form suggesting "E"

CSG models are constructed from primitives that are defined by inclusion of all points within the solid, *e.g.* the criterion for membership of a solid sphere is expressed as $|x-c| \leq r$, assigning specific values to **c** and **r** determines a specific sphere. Planar faced objects are defined by use of infinite half spaces. The primitives of CSG are then assembled by Boolean operations.

Boundary representation (B-Rep) is more related to wireframe than CSG. B-Rep models are based on face-edge-vertex graphs with data structures for surface geometry, curve geometry, and coordinates: analogous to that for the wireframe. That structure is usually represented by the winged edge topological structure face-edge, edge-curve, vertex-point shown in *fig. 4.4*. Surfaces do not have to be

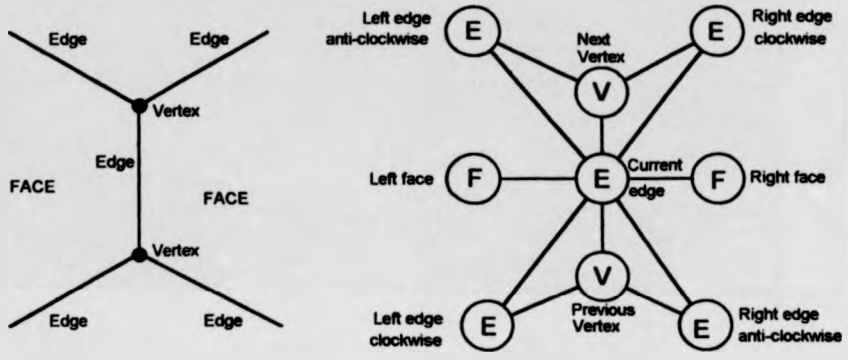


Fig 4.4 Winged Edge data structure for B-Rep
[Rooney & Steadman, 1987]

defined by analytic half-spaces, B-rep modellers can use a wide range of surface descriptions including free form or sculptured surfaces on shapes of limited extent. See for example [Woo, 1985]. The basic relationship on the face-edge-point graph is the Euler-Poincaré formula

$$V - E + F - H = 2(M - G)$$

where an object has V vertices, E edges, F faces, H hole loops (complete intersection of a section on a face, such as a cylinder creates a circular hole loop), M disjoint pieces, and G 'handles' or through holes. Faces in graphs obeying that formula may be multiply connected. The formula, having six variables all of which are integers, defines a 6-dimensional integral grid . A change of any variable (called an Euler operation) may be represented as a transition between points on that 6D grid. Thus any solid may be built up from single Euler operations. In practice these operations are combined into user-friendly groups to do things like Boolean operations, sweeping and swinging, and various tweaking operations.

Transforming from wireframe to B-Rep is fairly straightforward as the topological structure of wireframe is a subset of B-Rep. However transforming CSG models to

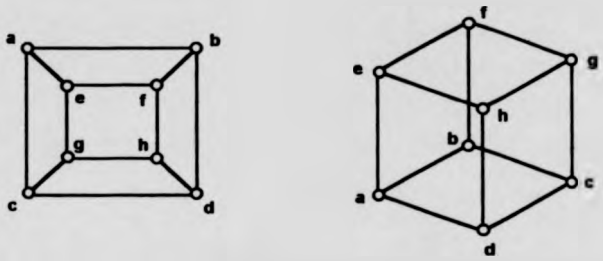
B-Rep or a common form is difficult since labels are very different and in general common vertices between the two representations must be derived. It is indeed an issue whether it is actually possible to have a transformation without additional information. It is also difficult to integrate the many different characteristics of geometrical models into a single unifying framework, a factor observed by [Smithers, 1987].

Definitive Shape Representation

If we specify shape sufficiently abstractly we should be able to produce a common frame of information from which many forms can be derived. One common feature of all shape definitions is the need for labels, whether to denote vertices, edges and surfaces, or to provide parameters to define analytic functions. A second need is to use graph theory [*c.f.* Wilson, 1987] to construct graphs that describe combinations of vertices on the one hand or connections between components on the other. We therefore build the algebra for our shape definition on labels and graphs. The shape model that was initially conceived was reported in [Beynon & Cartwright, 1989] but has seen some development since then. In essence, the underlying algebra incorporates three distinct sorts for describing geometric objects: complex, frame and object.

A **complex** comprises a list of labels, collected into a list of subsets of the set of labels. This is the normal way of representing abstract graphs. Labels may be thought of as nodes on a graph structure with the edge structure specified by the list of subsets. Alternatively they can be viewed as references to abstract points that lie in a Euclidean space of dimension $d \geq 1$; this means in particular that labels may refer simply to abstract scalars. The non-negative integer d is the dimension of the complex although that is not specified as part of the data type. The complex interpreted topologically is designed to capture the combinatorial ingredients of the object: reference points, dimensions and incidence information expressing the way the object is synthesised from simpler components. There is no coordinate information in a complex.

The complex allows us to represent structural information concerning geometrical objects in a graph form for B-Rep and Wire-frame model or to relate analytic variables in the CSG or indeed to cover the relationship of components in a multiple-feature object. It incidentally allows us to specify families of objects or assemblies with the same abstract structure (isomorphism). *Fig 4.5* illustrates two



BOX = ((a b) (b c) (c d) (d a) (e f) (f g) (g h) (h e) (a e) (b f) (c g) (d h))

Fig 4.5 Topological representations of the complex BOX

ways of showing a complex BOX on the labels a .. h with the same subset structure. The 2D representations convey very different geometrical shapes to the beholder.

One way to give geometrical substance to a complex is to link labels with locations in space. For this we use the **frame** that consists of a complex together with a list of coordinate vectors all of the same dimension d, whose role is to supply locations for the vertices. Even in this form a frame remains an abstraction from an **object**, though there is a canonical way to realise a frame as an object, the intention is that a frame supplies a finite set of parameters that can be used to represent essential reference and construction points from which an object may be synthesised. For example the primitives in CSG may be represented by frames with default coordinates: a block would be synthesised from coordinates corresponding to a unit sided box realised in straight line segments.

The **object** is specified not by its extent (*i.e.* the set of points deemed to be within the object) alone, but by the ingredients from which the extent is in general determined, obtained from the complex and frame information. Other information may be added to give the object position and shape in terms of scaling and isometries (where it is to be located in world coordinates and what scale factors apply in each axis). That would be similar to what one might do with an object imported into a drawing in a CAD system. An object may be defined on a single frame; in general it is determined by a list of frames, together with a function that takes the parameters of these frames as arguments and returns the extent of the object. Thus the combination of the whole set of frames constitutes the object. By further functions that state how the structure is to be expressed graphically we can instantiate the object for display purposes

4.23 Operations on Definitive Shape Types

The idea behind the choice of sorts (complex, frame and object) is that an object is to be viewed as the realisation of a combinatorial structure, as represented by an underlying list of complexes. Effectively we can use these sorts to represent "parametrised objects", although these will be more generic than ordinary parametrics.

Given the three basic sorts described we can create **types** to describe each. That will enable us to make suitable **operators** on these sorts. For example we may wish to have operators to synthesise new complexes from old, geometric operators that assist in the specification of coordinates for frames, or operators that combine objects as in CSG, or to specify objects, frames and complexes in terms of each other.

Since a *complex* is built up of labels, which are atoms or strings, the algebra will include operators for list and string processing respectively. From a list of labels list operations can be used to build standard subsets such as that shown in *fig.4.5* for constructing a box. We can combine complexes by set operations such as union and intersection or perform graph property extraction such as lattice properties, paths and cycles.

The *frame* associates coordinate information with the vertex labels, so operators may be classified according to their effect on sorts.

(1) Constructors and selectors able to construct and unpick frames
(2) Operators for accepting complexes and lists of reals to realise a frame
(3) Operators on frames for making new frames
(4) Operators on frames that return an object

Under (3) we can have all the ordinary vector operations (addition, subtractions, dot and cross products, isometries, scaling and shearing). More generally we might want operations such as forming the complex hull of vertices on their coordinates, finding the mid point of point pairs, building the union of two frames or a frame comprising the boundary of several frames.

Operators under (4) deal with realisation of objects. Edges may be realised as line segments, arcs or splines. (*c.f. Fig. 4.6*). A spline, for instance, is determined by a wire-frame together with an appropriate set of boundary elements. The wire frame has two ingredients: a combinatorial structure, consisting of an array of labelled

points, and an associated array of coordinates. By specifying how the spline is abstractly defined in terms of the frame and the boundary elements, without regard for their specific coordinates and scalar values, the spline can be specified as an abstract object. By subsequently supplying parameters for an appropriate function, *i.e.* specifying a suitable explicit list of frames, a spline is derived as an explicit object.

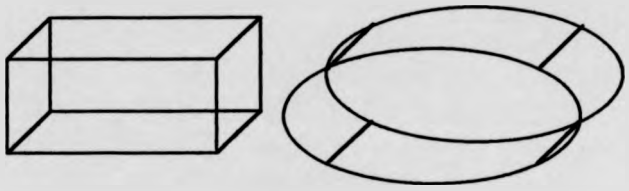


Fig. 4.6 Two realisations from the same combinatorial structure

Some of the operators attached to objects are associated with display representations. It may make little difference to the display picture whether the object appears as a 'polyline' rather than line segments, but it makes the possibilities of other operations for manipulation very different.

The representation of shape and operations have clear implementation dependent issues and are considered again in chapter 5 in connection with EdenLisp. Meanwhile we need to consider more profound issues in relation to interaction and the advantages that definitive notations introduce.

4.3 Extending Interaction

Interaction can be thought of in a number of ways. In computation the user and computer interact via inputs and outputs. In a physical assembly, such as a mechanism, the components interact via interfaces such as connections and bearings. Analogies can be made between these different notions of interaction though agent oriented ideas. The components of a mechanism are agents with a self behaviour and an interface with the rest of the world. In the computational sense we can extend the idea of 'interaction with the user' to 'interaction with agents', some agents being human, others being autonomously acting computational objects. This is the basis of the thinking in agent oriented programming in the ADM described earlier.

Before dealing with agents we need identify who or what are the agents in a designed object or system, or in the design process itself. We begin by following

up the discussion in chapter 2 on the design process. According to that discussion, design is best done by breaking the problem into relatively independent sub-problems in a hierarchical way. That enables us not only to discover the functions and shapes of components but also how they link with one another. We can then define components so that they can be dealt with as independent entities while not considering interfaces with other components. That provides one way of identifying those aspects of a design that can be accorded to one agent. We can then consider the nature of possible interfaces with other components to see how agents interact. Thus the first way of extending interaction is to be able to identify and structure potential agent elements in the design.

A second, related, issue has to do with constraints placed upon the user's interaction with the computer, for example in the way that a design is displayed and input is made. It can be awkward if the display can only show aspects of the design in a predetermined way. This is an issue on conventional CAD systems where the user interface, usually via a screen display, has elements of format that cannot be changed very easily by the user.

The third aspect of interaction is to do with changing state in a sequential way, such as in animation. We may wish, for example, show a series of positions in a locus as emulation of a moving mechanism.

These issues are addressed in the Abstract Definitive Machine (ADM) outlined in section 3.42 above. In order to develop the methods for design purposes it was felt necessary to examine the advantages and deficiencies of DONALD in particular, given that this notation is nearest to CAD.

4.31 Hierarchies in Design

The concept of environment is attractive from the point of view of design. We should like to be able to structure a design problem hierarchically so that the sub-problems can be put to one side for later development. DONALD has the ability to create that hierarchy though its notation

```
openshape ...
  within ... {
```

However the interactive element has some disadvantages. Consider the following script of definitions written by the author in DONALD for illustrating Bow's Notation for beams in bending. *Fig. 4.7* shows a sample output from Bow DONALD. A beam

is modelled from straight lines using the notation described in §3.4. The structure of the program is as follows.

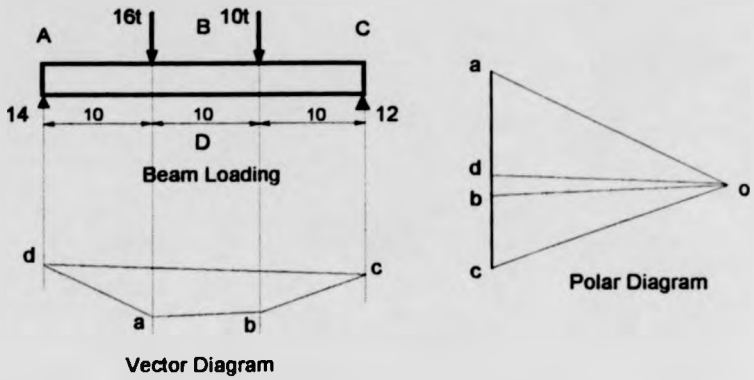


Fig. 4.7 Bow's Notation Method

Bow.DoNaLD Features
  Beam Loading diagram
    beam, load arrows, reaction supports, dimensions, leader lines
    labels: Bow's notation
    Definitions: position of beam, position of loads, scale factors
  Polar Diagram
    Position of Pole,
    length scale for loads,
    position of load vectors
    Labelling of forces **ab bc** and Pole **O**
    Lines from **O** to **a**: **Oa** and **Ob, Oc**
  Vector diagram
    Extension of lines from Beam Loading diagram
    Construction method:
    Copy polar line **Oa** and place in region **A** between the vertical extension lines from the end support and 16t load: trim to fit. Label as **da**.
    Copy polar line **Ob** and place in region **B** at the end of **da**; trim to form vector **ab**. Repeat for **Oc** in region **C** to form vector **bc**.
    The vector joining **c** to **d** is the required resultant.
  Result
    Copy the resultant vector **cd** back to the polar diagram as indicated in the *animation* with end **d** at the Pole point. Where it intersects the vertical line **abc** is the location of **d** on the polar diagram. **cd** and **da** are the magnitudes and directions (i.e. **c** to **d**) of the support reactions

The following is a segment of the DoNaLD code from the program script Bow DoNaLD that illustrates some of the points under discussion

```
### Bow's Notation for Beams in bending
point A, B                              # End points of the beam
line  AB                                # the beam
```

```
int   scx, scy                          # scale factors for the display
real  LB                                # beam length

AB  = [A, B]                            # define line with A and B ends
B   = A + [LB,0]*scx                    # position of B
scx = 20,  scy = 1                      # default scale factors
A   = (55,800)                          # origin of display beam
LB  = 30.0                              # default beam length

openshape forces                        ## beam forces and reactions
within forces {
  point C, D                            # Points of action of forces
  line RA, FC, FD, RB                   # Lines of action of
                                        #     forces and reactions
  real Ra, Fc, Fd, Rb, LC, LD           # Labels for
                                        #  reactions/forces/distances

  Rb = -(Fc*LC + Fd*LD) div ~/LB
  Ra = -(Rb + Fc + Fd)
  RA = [~/A,~/A+[0,~/scy]*Ra]           # Draw lines proportional
  FC = [C,C+[0,~/scy]*Fc]               #    to force magnitude
  FD = [D,D+[0,~/scy]*FD]
  RB = [~/B,~/B+[0,/scy]*Rb]

  C  = ~/A + [LC,0]*~/scx
  D  = ~/A + [LD,0]*~/scx

  LC = 10.0, LD = 20.0                  # Values for force positions
  Fc = -16.0, Fd = -10.0                # Values of forces
}
```

Addressing the first of the interaction issues: in BOW.DONALD we divide the problem. The polar diagram, beam elements diagram and force vector drawing are, for drawing purposes, separate problems and have their own definitions. For example, the code above shows definitions for the beam forces and reactions. The interlinking of the sub-problems requires access to local variables within the drawings In DONALD we localise variables using *within shape*. Access to variables is by having a labelling method that carries the address as well as the name of the appropriate variable. The scaling factors used in the display need to be common for the vector diagram to look correct. Access to those factors is by the symbol **~/**, a method reminiscent of the directory structure of Unix or DOS, *i.e.* **scx** is accessed by the name **~/scx** just as Unix directories address parent directories. In an OOP system that kind of addressing is difficult as interfaces need to be anticipated in order for messages to be enabled. New ways of linking are also impossible in OOP without accessing the code directly - and that would be against the spirit of information hiding.

Once entities are written interaction may be done directly by editing the definitions at the keyboard. Also we can use previously stored definition sets. Various entities

in Bow.DoNaLD have to be defined for routine components such as arrow heads and labels. These can be 'library definitions' easily adapted to the case in hand. It is worth noting that labelling is a particularly powerful aspect of the definitive interaction. Stored or explicit strings can be attached to labels and displayed according to the values of their definitions so it is very easy to link both the content and position of a label with any other entity. That makes updating the display extremely easy: labels automatically get moved to new positions and/or get new strings without further definition each time any item is changed. Indeed we have shown that it is possible to arrange that the labels are always positioned where the fewest line intersections with the text occur. Labels can also have values that change according to other interactions, *e.g.* constraints can have warning labels whose value is a string that contains information appertaining to the interaction just performed. These kinds of facilities lift the restrictions noted in Design View and illustrate the potential for definitive methods.

4.32 Indirect Interaction

The second issue is mainly to do with the presentation of the current design to the user. The set of definitions that form an object represents the current state of the interaction very effectively as the user can readily determine the current state of the dialogue at any time, and can predict the effect of any dialogue actions. However, the use of definitions can also be unnatural, since it requires an acyclic system of functional dependencies between variables. If the criterion for a good representation of the state of an interaction is 'predictability of response to dialogue actions', the restriction to acyclic systems of functional dependencies is superficially unnecessary. For example in DoNaLD a square, defined by points `a`, `b`, `c` and `d` where

```
b = a + [0,1], c = a + [1,1],  d = a + [1,0],
```

has dependencies on `a` that allow the square to be translated by a simple redefinition of the value of `a`. However those dependencies are not obvious from inspection of the graphical display and suggest that it may be better to think of the definitive notation as an intermediate code that can be automatically updated by the computer under certain circumstances. The definition of `a` could be changed by the computer if the user used a mouse or similar to drag the square. In that sense we could hide the value of `a` from the user (although, unlike OOP, that definition and value would be accessible if wanted). That device is in the Abstract Machine (ADM) where we called it an *action*. The action would be to update the definition of `a` depending on the current state of the user interface.

The idea that interaction from the user interface causes indirect redefinition can be extended. There is no reason in principle why the user interface itself should not be controlled by definitive methods. The current representation on a display should equally deal with the idea - its shape and state - and its current description in terms of what windows are open, what menu options are available and what forms of responses are required to inputs. With such complex tasks the computer is better able to cope than the user with the tedious task of redefinition. Given an appropriate method of updating (the *guards* and *actions* of the ADM) we can envisage that the state of the interaction can be changed by triggered actions.

4.33 Animation

To show, for example, the vector **cd** in *fig 4.7* 'move' from the force vector diagram to the Polar diagram we need to draw the vector in a number of intermediate positions. If that is done by redefinition there is no need to worry about deleting the entities when the next position is defined as the display is automatically updated when a redefinition is accepted. However the action of creating intermediate values is strictly a procedural one. DONALD shares the problem of PADL in that it is not possible to include procedural actions. Thus for interaction of this first type we are forced either into recording the intermediate positions as explicit definitions, or going beneath DONALD to the definitive interpreter. This is a serious disadvantage and ways of dealing with time related issues such as these are still a problem.

The definitive interpreter EDEN, an 'evaluator for definitive notations' is based upon a mixed programming paradigm [Beynon & Yung, 1988]. The EDEN interpreter has built-in support for a definitive notation based upon list processing, but can also be programmed to perform traditional procedural actions that may be synchronised with changes in the dialogue state using triggering mechanisms resembling those used in OOP. By translating definitions into the internal definitive notation it is easy to represent the state of the dialogue over any definitive notation. By using triggered actions it is easy to make responses contingent upon the current state of the dialogue. In effect EDEN makes it possible to link complex procedural actions and intricate systems of definitions: a very powerful programming paradigm but one that can prove difficult to use and analyse. It is possible to program directly in EDEN from within DONALD in order to carry out the procedural tasks described. In an experimental system that is permissible although a 'proper' notation would of course have those tools built in. Experiments in

'mixed' environments have been most revealing in the investigation of state. We turn to one of these experiments now.

4.4 A Computational Experiment

4.41 Background

The issues raised in the last section and in the implementation of the Abstract Definitive Machine led us to carry out an important experiment. In design work it is common to have to write short programs to solve particular problems, or to employ a commercial package such as a spreadsheet or MathCAD. For this experiment I took such a program, written in Pascal, to investigate precisely what interactions and states were implicit there. The program was designed to solve shaft deflection problems frequently required in machine design, presenting the output in graphs similar to those shown in the diagram shown below, *Fig 4.8.*

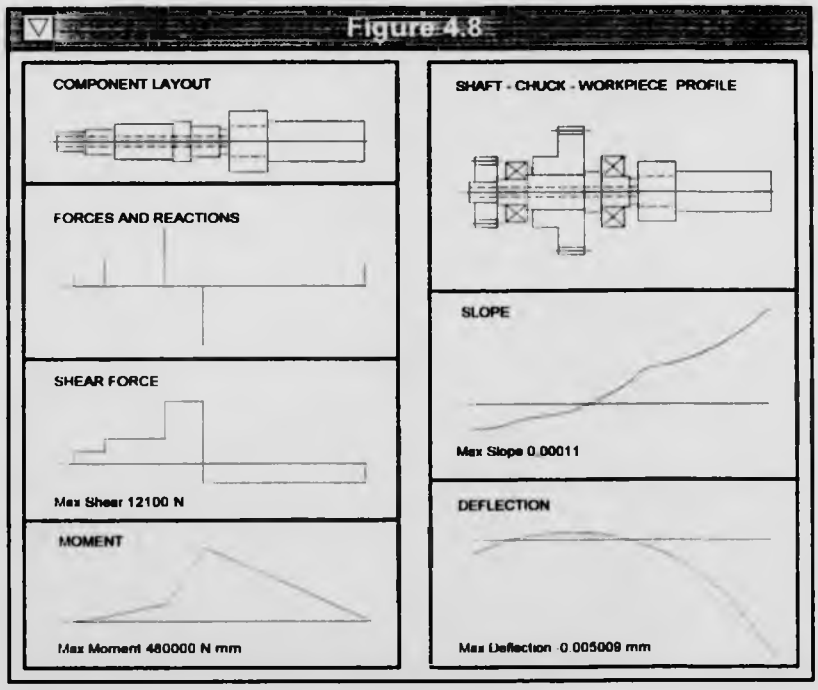


Fig. 4.8 Screen from Prototype Design Environment

The input allowed for a number of bearings and loading conditions and configurations, but the form of the output was always the same and the method of

describing the solution was fixed into the procedures. Rewriting the program into DoNaLD revealed that the hierarchies assumed in the Pascal program were designed to carry out the computation of specific problems required by the output rather than modelling the problem in terms of abstractions of shaft arrangements, about which information could be obtained for particular cases. The input shaft was divided into segments of constant cross-section and the section properties under stress were computed by the transfer matrix method. Computation involved several independent trawls of the input in order to set up the matrices and solve them for deflection and other properties. Procedures were then set up to enable the sequence of interactions required for articulating the design.

The hierarchies for a proper modelling of the shaft should be designed to show up the functionalities and features that may at any time be altered so that any kind of experiment may be performed on the model. For example we may wish

- to devise a representation of the drive shaft/workpiece configuration from a set of parameters
- to display engineering data for different configurations
- to experiment with geometry, *e.g.* additional segments or tapered segments
- to vary the disposition of components
- to monitor the deflection as a function of manufacturing cost
- to simulate the lathe in use
- to adapt the system to new uses such considerations of strength or appearance

4.42 Method of Approach

We can simplify the tasks by applying a definitive method with its intrinsic ability to deal with 'what if?' mode of analysis. A 'what if?' scenario is defined by a state and a set of latent transformations of that state: it expresses our expectations about how the various ingredients of the shaft model are interrelated. Those relationships are very different from those used in the procedural approach as part of the hierarchical structure in *fig. 4.9* shows. However, segmenting the shaft and isolating the relationships that are peculiar to each segment or ends of segments show that it is possible to get at and use variables such as segment length, material density and section modulus in order to compute cost properties.

We can extend that structure in all kinds of ways. For instance we can model the fact that if the dimensions or location of the gears on the lathe are changed that

will affect the distribution of load and alter the engineering data on display. Warning texts can be values of labels related to constraints.

Modelling fundamental data dependencies enables us to represent information about the design object so that it can be appreciated through experiment. The dependencies reflect the essential nature of the object as we choose to observe it. Their representation does not commit us to a particular strategy for transforming or interacting with the design object; it models those aspects of the observed behaviour of the design object that we wish to take for granted. This makes it much easier to describe procedural activities associated with the design process, such as enhancing the design model, developing the design environment, or simulating the design object.

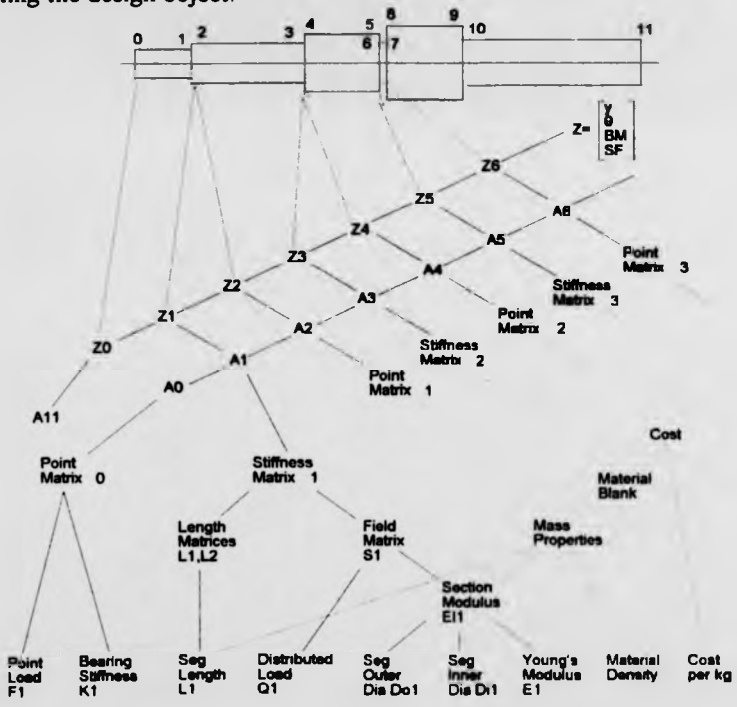


Fig. 4.9 Hierarchical Relationships in Shaft Analysis.

4.43 Implementation

The Implementation of 'shaft' was attempted in the philosophy of the ADM described in §3.42. The hierarchies described above may be defined with sets of definitions. The difficulty is implementing the way that actions can be done, as was

noted in trying to construct Bow.Donald. Rather than use EDEN the idea of a *script* was developed. A script of definitions corresponds to the STORE 'D' in *fig. 3.2* as part of the ADM model. The whole shaft analysis display shown in *Fig. 4.8* is described by a script of definitions specifying relationships between windows, geometric components and textual annotations. Many kinds of interaction with the computer model correspond to simple redefinitions or extensions of the script. An important feature is that each interaction with the system leads to an incremental or wholesale change in the script: either to enrich the design model or to add functionality to the design environment. The significance of design environments will be examined in chapter 5. In this context the important feature is the ability to change the pattern of the script itself by actions.

The problem of implementing actions is apparent if we attempt to add further shaft segments. It will then be necessary to make wholesale change to the definitions that control the graphs. Sections of the script will have to be rewritten with new points, lines and shapes being declared and defined. This is a fearsome task to do manually and ought to be hidden. One way to illustrate that re-writing is to carry it out explicitly. A high level text generation program was implemented in EDEN to create the new script that is to replace that the previous state. The action in the original state triggers a call to the text generation program with the data required for the new definitions. The new script is then generated and passed to EDEN interpreter to be implemented. Since this is a form of self modifying code (using EDEN to create EDEN code that is then passed back to EDEN) the process was carried out by means of separate files that were piped in the appropriate way. This rather clumsy method showed that the principle of the ADM could work. It remained for developments described in the following chapters to see how a cleaner environment for actions might be structured.

4.44 Results

We have seen that the computer can act as an agent in the process of design by changing a script of definitions to describe a new state. It is not difficult to see that different actions in the original script can cause different scripts to be produced. In that sense a set of definitions plus actions has not only a computational state but the ability to move to an infinite number of other states according to the combination of actions that is triggered. If we now structure the scripts themselves into the different aspects of the shaft design we can discern the following aspects.

- The topology and geometry of the shaft
- The database of relationships on the components and shaft mechanics

- The user interface: ways of accessing the prototype: mouse, keyboard, disc, etc.
- The display or output device showing some representation of the prototype

These aspects are illustrated in *fig. 4.10*.
Using the arrangement a number of trial designs illustrated the versatility of the script:

- to relocate or change the dimensions of components on the shaft, redefine a parameter; the corresponding distribution of load will be recomputed automatically and engineering data updated.

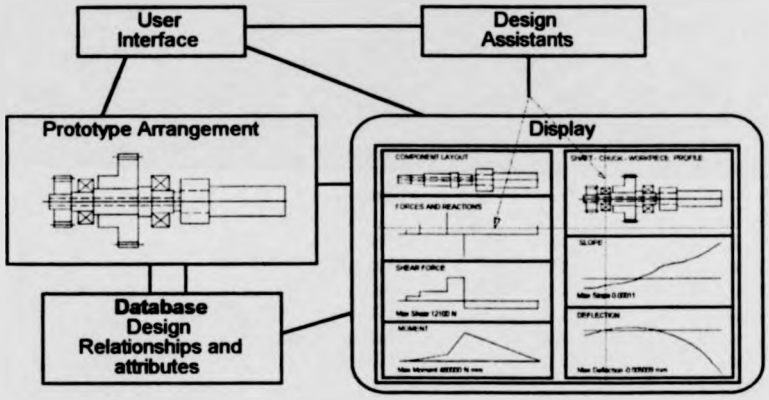


Fig. 4.10 Components of the Shaft Design

- to rearrange the display, or to relate window locations, redefine the opposite corners of appropriate windows
- to take account of a through bore in the shaft introduce a definition for the inner radius and redefine the function for second moment of area
- to monitor the effect of changing bore size, set up a textual window with a warning message that is displayed if the bore exceeds a critical size and is empty otherwise; then redefine the bore.
- to introduce a sweep line or pointer on the diagram define a line or shape whose location is determined by a sweep parameter or pointer identifier. Such a line can be regarded as a "design assistant" enabling individual components of the display to be examined in more detail.

The important feature of these examples is the power of redefinition. Using that ability to reprogram on the fly enables one to consider the whole design environment as a part of the design process. As the model maker sets up the model

materials, scale and environment for a physical prototype, so the designer should be able to set up an appropriate design environment in a computational model.

A issue that was explored briefly is that scripts within these groups can be considered to be independent and interactions between them would be the subject of structured interchange. A change made on the display would trigger changes on the object script. Conversely a textually based redefinition of a bore diameter would be reflected in the display of the shaft. Different parts of the design may be considered to be agents in the design process.

For further details of the work on shafts see [Cartwright & Beynon, 1992].

5

A Design Prototyping System

5.1 Design Process Support

5.11 Specification

In the consideration of the design process the main criticism levelled against computational support for conceptual design is that 'content' outstrips form, so any preconceived structure on design tends to over-constrain that part of the design process. In an ideal support system the designer must be free not only to create the product in mind in as free a way as possible but also to be able to tailor the working environment similarly. Veerkamp, in probably the best exposition of these requirements, makes clear the relationship between the designer and the design, emphasising that the designer must be able to exercise his (*sic*) faculties.

> 'I believe that the essential thing in designing is that the designer creates his own design environment and the system must give him the freedom to do so. The system must understand the designer's commands and translate them into system tasks.
>
> ...
>
> The designer and not the system determines the way the design process is directed.
>
> The user interface must allow the designer to express his ideas in his own terminology" [Veerkamp, 1992]

Some reasons for needing control of the working environment were mentioned in the last chapter. The designer interacts with the system in order to change the state of the design. However if the system merely records descriptions of the design then a significant change in the design may involve considerable adjustment to that description. If the adjustment must be done explicitly by the user there is a serious

risk of error, but if it is looked after by the computer constraints may be invoked that are functions of the support system. We have already seen how geometrical modellers are constrained in that way.

So what do we want in a support system? [Tomiyama and ten Hagen, 1987] and others say that a design support system should be:

1. a place to describe what designers have in mind
2. a tool to verify the designer's ideas in terms of feasibility, cost, performance, *etc.*
3. a system to store and retrieve design information and to transform it amongst various subsystems.

I would put the first of these rather differently in the light of the forgoing discussion. The system would be more useful if it not only described the object but that:

- The model of the design is as faithful to the physical world as possible
- It has relationships based upon observation and experiment on physical objects
- One is able to experiment on the model as one might do on a physical model

Tools for modelling the designer's ideas need great flexibility. Any convenient geometrical modelling method should be allowable, preferably interchangeably and in combination (*e.g.* CSG, B-rep, wireframe). That means addressing the problem of transforming between models that may contain or be based on very different descriptions of designers' ideas

Interaction is crucial, particularly in the evolution of the design. At the design level it should enable

- *variational design :*
  given a value for a design attribute, the system should respond by
  - checking constraints
  - propagating the effect through the design
  - monitoring : to see if more changes are needed to specify the design
    to show a suitable message if a constraint is violated
    to show up conflicting constraints or design decisions.
- *design history*
  - exploring "What if ....." questions
  - able to return to an earlier design

At the user interface the screen arrangement, the input tools and the image on the display should all be definable by the user. Design assistants in the form of pointers, image tracking, moving crosshairs and so on should be definable rather than pre-conceived

Integration is usually interpreted as the ability to communicate in a consistent way with application programs or macros external to the system. In that context it is worth noting that there are two kinds of integration. Designers may embrace the development of the entire product from conception to manufacture, or they may specialise in one special phase of design for many products. Historically the latter has tended to be the practice in most countries (although, interestingly not in Sweden). [Wærn, 1986]. Where the designer is concerned with only a single phase then design support may centre on areas of specialisation, *e.g.* customisation of the CAD system, access to data bases, and linkage with analytical methods such as finite element analysis. On the other hand, integration of all product activities is becoming more common with the development of design groups or 'forward engineering' noted in section 2.5 above. With design groups more sophisticated management and development of the computer model are necessary to cope with many different agents of change. The support system should allow the concurrency of different users simultaneously interacting, or from computational agents looking after internal consistencies arising out of user actions. An integrated system also requires the ability to link different disciplines, for example electronics, chemistry, precision engineering linking downstream with manufacture, assembly and servicing.

5.12 Approaches to Process Support

Knowledge Based Systems

The historical tendency has been to construct Design support tools to aid product description rather than the design process. The design process is abstract, but knowledge about the product and its evolving description may be structured and managed in various ways; *e.g.* help can be provided to the designer in organising the thinking required, suggestions and starting points can be provided though browsing in libraries of appropriate entities. The Edinburgh Design Support System [Popplestone, 1984, Popplestone, Smithers *et al*, 1986] is an example of a knowledge based system. It uses an "Encyclopaedia" of engineering knowledge, with a method for creating a "Design Description Document" that contains knowledge appropriate to a particular design activity. Popplestone, *et al*, identified the basic requirement for retaining design knowledge in a consistent way,

particularly where a system has to operate in many disciplines as the design process continues. Process support may be assured by remembering the generated knowledge in a structured way so that later stages will be automatically constrained by earlier knowledge of the process. The difficulty here is when significant changes are requested at a late stage in the design process as fairly massive reworking will be necessary to update the relationships.

Feature Editors

In feature editors the design process is circumscribed in a preconceived way. The constraining effects of feature based systems are particularly serious when the bases of features are manufacturing methods. By definition particular manufacturing techniques can only produce a finite family of features, albeit infinitely associative. For example a feature editor based on manufacturing methods reported by [van Houten & van t'Erve, 1992] coerces the design process to be essentially design for manufacture. In [Case, 1992] the author comprehends feature technology as an integrating methodology between CAD and CAM, but he admits that the number of features that can be recognised is limited and the designer's intent may be lost. Even so, he believes that it may be worth working within those constraints if gains in productivity are realised.

Object Oriented Programming (OOP)

Significant attempts at supporting the design process have come out of the OOP stable. The important point in the philosophy of OOP is that the user is free to model reality without having to go into the mechanics of how the model itself works (i.e. how it is programmed). That forms the background to the development of design support systems using OOP. The designer can concentrate upon the objectives of the design in hand to develop a particular product. Rossignac, *et al*, [Rossignac, Borel & Nackman 1989] identify the need for the designer to be in charge of the design process, claiming that it is up to the user to come up with an operation order that will meet the functional requirements of the required system. The system can then support the design process by providing an intelligent apprentice, able to deal with well-defined tasks encountered during that process.

Mäntylä follows the accepted notions of top-down or strategic decision-making and bottom-up detailing for the conceptual and detail design stages, [Mäntylä, 1990]. He supports the need to focus on particular aspects of the design and to capture the sequence of focus changes. He highlights the need for capturing and preserving design history and alternative elaborations from a common starting

point. Although he argues for tackling the problem of process modelling, Mäntylä's Browsers still make the designer responsible for using the system in a structured way.

Reasoning more deeply about the design process, Veerkamp suggests that the designer thinks in a *goal-oriented* way, [Veerkamp, 1992]. He distinguishes meta-level reasoning and object-level reasoning. The former is concerned with strategic decisions on how to proceed with the design - what must be done next - and with the formulation of design goals. The latter is concerned with the current state of the design object - how to extend the model to reach the current design goal. The subdivision of such goals describing the process is a tree structure with meta-level goals as nodes and object-level goals as leaves. He concludes that it is possible to construct a descriptive model of the design process.

Veerkamp's Artefact and Design Description Language (ADDL) is designed to be interactive, working with a data-base called the fact base. The fact base contains all the literal facts currently known about the object and is extended as the design proceeds to contain more and more detailed information about the artefact, describing the structure of parts of the overall objects. The object base, a sub-set of the fact base, stores these parts as separate objects, each with its own state. The system operates on these data, via the Interpreter, with the active co-operation of the designer, via the user interface. The Interpreter uses *scenarios*. A scenario is a piece of design knowledge employed by the system to perform a design step. It consists of a set of methods and rules that query the object information state at a particular stage of the design process, depending upon the current design goal. A scenario, once activated will continue to be interpreted until it terminates. At that point the state of either the design process or the design object is updated, and a new scenario is chosen. A sequence of activated scenarios represents the design process. Back tracking allows poor design directions to be ignored and earlier stages to be points for restarting the design process.

Provision for process modelling in ADDL is made in that design objects may be modelled independently of certain contexts by means of the meta-model mechanism. Assumptions generated at the object level may be transformed to process parameters in the process information state, the method being step-wise refinement from an incomplete to a detailed description. A single design step is performed by means of a scenario. For each incomplete state of the design object a

scenario appropriate to that object is selected and executed. The design knowledge is represented as rules with a forward chaining strategy for inference

The ADDL suggested by Veerkamp give good support for the design process, but again the process is descriptive and static. Changes have to be propagated via the scenarios that encapsulate information that one may wish to access.

5.13 Agent Oriented Approaches

The agent concept is central to the way we deal with the design process. It can be interpreted in different ways according to which aspect of the design process we wish to address.

First, the components of an engineering system can themselves be regarded as agents. By analysing the relationship between them we represent knowledge about how the behaviour of the entire system is related to the components.

Second, different experts acting on the same design product are agents. An engineer may redistribute elements in a design without regard to their aesthetic appearance or cost of manufacture. A manufacturing engineer might examine features in terms of ease of manufacture without reference to the design functionality. By analysing their interrelationships we understand how to represent the design to the different participants and to identify potential conflicts between their requirements

Third, the computational elements required to construct a computer model of the engineering system and to simulate its behaviour can be regarded as agents. By representing them we both record the current state of the design and prototype the system to be constructed in software in such a way that we can perform realistic experiments on the computational model.

One approach to using agents is to consider the user as a kind of super-agent in charge of a series of autonomous computational agents. That generalises the idea of packaging independent computer programs such as the I-DEAS, CADAM, and ComputerVision systems. In commercial systems the "integration" is done by internal translation but the user is responsible for guiding the product through the separate stages. In a more agent-oriented approach the autonomous packages have specific tasks around the product design but the modelling methods are linked at a high level. The first of that type is that being developed by Tomiyama's group in

Tokyo, based upon the ideas discussed in chapter 3 above. The latest developments of that are described in [Tomiyama, *et al*, 1994]. The core of his framework is the Metamodel system that manages the different models used by the functional modelling and external analysis programs so as to link appropriate model concepts for use in later stages. The knowledge management system also links different kinds of knowledge: catalogue, physical, contextual, abstract, assumed or observed knowledge.

There are a number of design systems being developed based on a framework of linked systems. One that explicitly explores the notion of agents is being developed at Lancaster University called Schemebuilder [Oh, Langdown, Sharpe, 1994]. The general idea of having different agents is that one may wish to carry out design tasks like simulation, component selection and layout. The principle is that of embracing a co-operative relationship between the man and machine, providing decision support that augments design creativity by guidance and suggestion rather than an expert system that tries to replace design functions. The agents are in fact a heterogeneous set of software systems (including MetaCard, based on Hypertext, a Knowledge Engineering Environment, Simulink and a CAD system) linked by a combination of Unix pipes and shared file mechanisms. The underlying philosophy follows [French, 1985] in positing that Conceptual design is a process of structured logical thinking whereas it is apparent from our earlier discussion that the process may be anything but logical. There is however some provision for browsing via hypertext that provides opportunities for experimentation.

Turning now to Agent Oriented Definitive methods the agent idea may be much more explicitly worked out using the ADM principles discussed earlier. If we link sets of scripts so that they can be modified according to the needs of the different agents who may wish to interact with the design, then we have conditions similar, for example, to the scenarios described by Veerkamp. There is an important difference though: the scripts will have linked actions according to their agent of origin. Each agent has its own script and privileges with respect to changing both its own script and those of others. The collection of all participating agents creates a system that effectively has state. The state of the interaction and possibilities of other states depend upon the actions triggered by redefinitions.

In Definitive methods the fundamental abstractions for constructing state transition models for complex interactive systems are outlined in principle in chapter 4 above and is reported in [Beynon *et al*, 1990, and Beynon & Cartwright, 1993].

We have seen the importance of the 'experiment'. In the Abstract Definitive Machine the interaction between an agent and its environment is modelled in terms of observations, recording the agent's view of the state of the environment. Observations are represented by variables whose values can be changed through actions by agents in the system. Observations are also of different kinds. They may be measurements that an engineer makes; they may be concerned with the behaviour of the system from the viewpoint of one of the agents. Alternatively they may be associated with computational agents and correspond to variables to which they respond or which they can conditionally change.

The design process involved in agent orientated modelling is directly analogous to that which is carried out with a physical prototype. Any changes in a prototype naturally have a knock on effect on the design. Move or lengthen a lever and the whole linkage is distorted. Change the shape of an object and its relationship with the environment (*e.g.* its wind resistance) is also affected. In simulation, define the state of the display in terms of internal variables and the display is automatically updated as they change.

The potential of agent oriented support of engineering design is greater than any previous approach because of its fundamental difference in the way that it handles state and state changes. However its practical outworking has proved to be a hard road.

5.2 Evolving a Prototyping system

5.21 Definitive Notations and EDEN

The Abstract Definitive Machine (ADM) and agent oriented programming have been slow to evolve. The idea of state and action came out of the first real Evaluator of Definitive Notations [Yung, 1987; Beynon & Yung, 1988]. EDEN is written in C under Unix and supports limited graphics, now under X-windows. EDEN allows a script of definitions to be evaluated much as a spreadsheet. More significantly it has built-in support for a definitive notation based upon list processing, and can be programmed to perform traditional procedural actions that may be synchronised with changes in the dialogue state using triggering mechanisms resembling those used in OOP.

To explain its operation we use the following dialogue with the EDEN interpreter. EDEN responses are written after the $ prompt

```
1.   x = 34
     $ 34
2.   p = writef (x)
     $ 34
3.   x = 9
     $ 9 9
```

Statements 1 and 3 are definitions in the conventional sense. Statement 1 assigns the integer **34** to the variable **x** and returns **34**. 3 redefines **x** to be the integer **9** returning **9**. Statement 2, however, defines a procedure, not having a 'value' in the strict sense. Following the input of the second statement the number **34** is displayed. After the third input the second definitive statement is invoked automatically again and because the argument of **p** has changed, the number **9** is displayed, hence the two responses of EDEN.

More subtle is a procedure that is triggered by a variable different from its arguments. The statements

```
4. b = sin(0.56)
        $ 0.5311861979209
5. p = writex : b (x)
        $ 9
6       b = 12.9
        $ 12.9 9
```

cause the procedure **p** to print the value of its argument **x** only if the variable **b** is changed. So after statement 6 **b** is redefined to have the value **12.9**, but also **p** is activated and prints **9**, the value of **x**. (**b** is simply the trigger: its value is irrelevant to the procedure in 5.)

Because it is a mixed programming environment we need to be careful to decide what is meant by the 'value' of a definition. As observed above we cannot say that the value of **p** is **x** since the definition of **p** is procedural and **x** is displayed by side effect, *i.e.* it does not change anything in the programming system or data. In EDEN we separate definitions-with-values such as statements 1,3 and 4 from procedures like 2 and 5, calling the first *definitions* and the second *actions*. In the ADM we go further in defining an action. There an action may do more than side effect. It may actually cause new definitions or redefinitions to be invoked. Possible ways of doing that are described below (section 5.4)

EDEN makes it possible to link complex procedural actions and intricate systems of definitions. It is a very powerful programming paradigm but one that can prove difficult to use and analyse. One way of using the paradigm is to treat EDEN as low-

level. Realistic programming tasks often require routine detailed sets of definitions, *e.g.* for graphical entities such as points and lines. These definitions can be automatically generated using a suitable translator of higher level definitions. That is the basis of the DoNaLD notation. DoNaLD statements define lines for instance by a type declaration and end points. The necessary detailed EDEN definitions and the actions to put lines on an X-windows display under Unix are complex but normally should be transparent to the user. (Unlike OOP however the information is not encapsulated. The EDEN code can be inspected. Indeed it is not difficult to write the code in EDEN ; it is simply tedious for routine work).

Several different notations that translate into EDEN have been devised by others in the research group, each designed to enable a particular specialisation. For example CADNO [Stidwell, 1989] is an attempt to represent more abstract topological ideas than DoNaLD and to produce more complex geometrical modelling tools. SCOUT is a notation that deals with the complexities of interfacing windowing systems.

By translating definitions into the EDEN interpreter one can represent the state of the dialogue over any definitive notation. The examples cited in chapter 4 illustrate how that works out in practice. However if we wish to implement the more comprehensive underlying algebra for dealing with shape representation expounded in that chapter there are practical problems. The graphical tools available under Unix are inadequate to deal with the geometrical modelling of objects described there (CSG, B-Rep and spline constructions) when compared with specialist CAD systems. Rather than re-invent the wheel I felt that the best way of proceeding with the prototype design support was to link a definitive notation with an existing CAD system. While that meant that the definitive notation would not be pure, in that it is not definitive "all the way down", EDEN itself is also not pure, being written in C and calling procedural routines for graphics and operating system actions. A bonus of linking with a CAD system is that it shows that the definitive approach is not exclusive and can easily interface with other systems.

5.22 Development of EdenLisp specification

The idea of a definitive notation interpreter linked with a CAD system was influenced not only by the constraints of the existing graphical tools but also by pragmatic problems of implementation. For an engineer the idea may be important but once accepted the gearing problem takes over. Gearing is the ratio of the time

to prove the idea, to the time to get a productive system operational. In many cases ideas never get into production because gearing is too great. I wanted to show that it is feasible to implement definitive methods so that their benefits are exposed and their commercial implications can be explored. A danger in that approach is that while research in the basic idea is going on there is a strong possibility that a particular implementation will be overtaken by events. I therefore needed a medium that would be adaptable to such changes. Since EDEN is essentially list orientated and declarative in style it made sense to use a list based language linked to a CAD system that could interpret it. The medium chosen was AutoLisp®, the interpretative language of AutoCAD®. [AutoDesk, 1987, 88, 90, 92]. AutoLisp is a superset of Lisp, consisting of a Lisp interpreter that accepts the common core of Lisp expressions together with all the AutoCAD commands, so enabling the interpretation of Lisp statements to be both computational and graphical.

The name EdenLisp was attached to the new notation to link it with its pedigree. The initial specification was for a definitive notation interpreter similar to EDEN. That meant creating a program in Lisp that could take an input in the form of definitions and carry out evaluations after the manner of a spreadsheet. Since Lisp is recursive in form and centres around lists the evaluator could be structured around trees. For example the script in fig 5.1 has the tree shown on the right.

```
1.  a=b+c
2.  b=f+2
3.  c=d+e+f
4.  e=1
5.  f=3
```

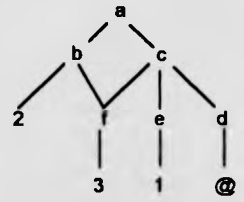


Fig 5.1 Tree form for evaluating definitions

Here `a` is related to `b` and `c`, `c` to `d`, `e`, `f` and `b` to `f` and `2`. `e` is related to `1`; `d` is undefined and given the symbol `@`. On receiving the 5th statement the evaluator would assign the value `3` to `f` and then inspect the branches of the tree above. Seeing two branches (to `b` and `c`) evaluation would proceed with each in turn: `b` would be re-evaluated by inspecting its leaves in turn. If all leaves have a value then `b` acquires a value, otherwise it keeps the undefined symbol `@`, in this case `b=5` is assigned. The same procedure applies evaluating `c`, except that if the value of `d` is undefined the value of `c` also remains undefined. Proceeding further up the tree from `b` and `c`, `a` acquires a new value if `b` and `c` both have values. In this case

`a` remains undefined. Thus after statement 5 the values of variables `(a,b,c,d,e,f)` are respectively `(0,5,0,0,1,3)`

While EdenLisp has the EDEN engine as its core, the basic philosophy of Lisp is such that higher levels of definitions can be created by defining higher layers of functions. The underlying algebra of the notation can be implemented directly by defining appropriate Lisp functions, rather than having a translator such as DoNaLD uses. That greatly simplifies the task of exploring new ideas. However before any algebra could be implemented it was necessary to have control of the input to EdenLisp to ensure that it is in the correct form. Input statements are not Lisp code and therefore have to be correct syntactically and semantically according to EdenLisp. A lexical analyser and parser were necessary to deal with those aspects respectively. Evaluation on the other hand is carried out by the Lisp interpreter by assuming that formulae are in Lisp. The right hand side of definitions and all commands have to be in the form of Lisp statements. Whilst that is not a problem for a Lisp programmer it is somewhat clumsy to write for example `(* a b (/ (+ c d) e ))` instead of the more conventional form `a*b*(c+d)/e`. An obvious addition to the parser was a translator that converted the conventional form into the Lisp form for internal computation.

The system should be able to deal with an algebra of abstract types for objects, and that requires that input is strongly typed. Typing enables us to have control of the way that an object is built up and simplifies the propagation of the computation. Lisp itself is untyped, although it will identify certain forms such as integer, real, string, atom, list, and symbol. Type declaration and checking, both on input scripts and internally generated scripts, had therefore to be part of the specification of EdenLisp. While that means that the user of EdenLisp must operate within a given set of types, anyone familiar with AutoLisp can add new types without difficulty.

The types specified for EdenLisp are intended to be those associated with the algebra outlined in chapter 4. However in its evolution EdenLisp has followed the path blazed by DoNaLD and CADNO. Emulations of those languages were constructed in order to test the system and develop the types necessary to describe topology and geometry. Models were developed in a similar way to that used in DoNaLD with the data structure and symbol manipulation under the control of EdenLisp. The AutoCAD system is called on when it comes to the display of geometrical models. It is used in two ways: to actually display information and also to obtain display data from the AutoCAD database for use in redisplay, or to delete

previously displayed but now superseded entities when a redefinition causes a display change.

The most revealing aspect of specifying a Prototyping system has been in dealing with actions. Lisp is extremely powerful in its ability to reproduce itself. Lisp code can be encapsulated within Lisp quite easily so that it is treated simply as a list. In that respect Lisp differs from languages such as C that has to use clumsy string-to-variable transforms. It makes the task of producing new scripts out of "actions" much more straightforward

5.3 Characteristics of EdenLisp

5.31 The Language

EdenLisp is defined syntactically by the grammar described in Appendix 1 and by lexical rules covering name construction and the use of the semi-colon and comma characters. Where alphabetic characters ("A".."Z") are allowed only uppercase are significant. They may be entered either as upper or lower case, but EdenLisp always translates them into uppercase. The following subsections describe the features of the language in an informal manner. Chapter 6 deals with the formal definitions and implementation issues.

Identifiers

EdenLisp users may specify identifier names for any variable. Valid user names are alphanumeric character strings that

1. begin with an alphabetic character ("A".."Z")
2. may contain any number of alphanumeric characters, or characters from [$,£,%,_] although names are stored in AutoLisp in batches of 6 characters, so longer names take longer to access.
3. are not identical with an AutoLisp, AutoCAD or EdenLisp linguistic terminal defined in Appendix 1. To do that will redefine those terminals with odd results. In particular the letter "T" is used in Lisp for "True" so if it gets redefined there is real trouble. (EdenLisp warns the user if a terminal name is declared)

Basic Data types

In the current EdenLisp the following data types are defined. The declaration symbols are alongside.

integer	INT
real	REAL
string	STR
list of integer	LINT
list of real	LREAL
list of string	LSTR
list of list of integer	LLINT
list of list of real	LLREAL
list of list of string	LLSTR
frame	FRAME

A further type available under Lisp is the quoted atom, parsed in EdenLisp as QATOM. The quoted atom is an extremely useful device. It consists of a Lisp atom (a single name or number) that is dealt with simply as an unchangeable object without being evaluated. The use of POINT in place of LREAL has been allowed and may be used interchangeably to help readability. The first three types in the table are the same as those in AutoLisp. The second three are flat lists each of which have members that are all of the appropriate type. The third three are flat lists of the respective second group. FRAME is a special type since it deals with lists of various types.

Statements are of three kinds:

1. *variable declaration* of the form

   ```
   type : username
   ```

   which sets up a variable of the given type with an undefined value or definition denoted by the cipher **0**. Further usernames of the same type can be declared by appending them to that statement with some white space between. For example

   ```
   int : a b c
   real : j k l
   str : stringA stringB
   ```

2. *definition statement* which may have of either of the forms

   ```
   username = <expression >
   username = if relational_expr then statement [else statement]
   ```

3. *commands* enclosed in parentheses. These may be to do with setting up object structures or they are AutoLisp functions that may invoke AutoCAD commands. For example

`(openwin 'Swindow)`
is similar to the DoNaLD statement: `"within ( ... "` .

Expressions within definition statements may be any of the following

constant

explicit constant *e.g.* `78, "word", (5 6)`
constant arithmetic expression; *e.g.* `56.8*log(0.67)`
symbol name or list of symbol names *e.g.* `[a, b, c]`

formula

algebraic *e.g.* `a*b/8`
defined function *e.g.* `Projn[LineA, 2]`

geometrical function

a function that actually represents geometry that may be displayed

Scripts are sequences of definition statements. Whilst a single definition statement may by arbitrarily long it is better to have a sequence of definitions. Scripts containing sequences of short definitions aid clarity and make for easy editing. They also make constraint enforcement and redefinition much easier. The following example of a script uses real expressions. (Line numbers are added for reference, they are not part of EdenLisp.)

```
          Monolith.Lsp
1.    real  : th R b L E Omax pi
2.    real  : F Mmax K Kt Qmax
3.    real  : thetam lamda TorSt

4.    thetam = 4*K*R*Omax / (Kt*E*th)
5.    lamda  = E*b*th*th*th / (24*K*R*L*L)
6.    Mmax   = b*th*th*Omax / (6*Kt)
7.    K      = 0.166 + (0.565 * th / R)
8.    Kt     = 0.325 + ((2.7*th)+(5.4*R))/(8*R + th))
9.    Qmax   = thetam*L

10.   E = 72000.0  ; aluminium
11.   b = 5.0
12.   th = 0.5
13.   R = 5.0
14.   L = 55.0
15.   Omax = 100.0
16.   F = 400.0
17.   th = 0.3
```

The example illustrates characteristics mentioned earlier.

1. Entities can be used before they are defined, *e.g.* `K` and `Kt`. This is a very powerful aid to top-down design.

2. We can change the order of the sequence without affecting the values, unless a redefinition changes an existing value. Line 17 redefines `th` from its definition in line 12 from 0.5 to 0.3. In that case EdenLisp coerces a unique definition by using only the last reference to `th` in the script: line 17.
3. Self referential or cyclic definitions are not allowed

Comments are allowed in EdenLisp after any definitive statement provided the comment is preceded by a semi-colon. A comment may be on a separate line. EdenLisp does not store comments when it reads in a line of input. When line 14 is read in, EdenLisp discards the comment and semi-colon: `; aluminium`

String Reconstruction

Definitive statements are not saved in string format by the interpreter but translated into Lisp and can only be read back from EdenLisp in that form. Commands are not reconstructed or saved, they are executed and then forgotten.

Real and Integer Expressions

1. Real and integer expressions may contain real sub expressions, real variables and constants and the usual real operators. Examples of all these are given in the program.
2. Unary minus "-" may precede any expression.
3. All the other AutoLisp functions relating to reals and integers listed in Appendix 1 are usable in EdenLisp.

Lists and Functions

Lists are fundamental both to EdenLisp and AutoLisp. However some restriction is placed on the format of lists so as to enable strong typing and type checking. Lists are therefore created using the special operator [ ] together with commas to separate the list entries. Both devices are foreign to Lisp since the latter simply uses parentheses ( ) with white space as separator. To create an EdenLisp definition as a list we write

```
a = [1,2,3,4]
```

Lists can of course contain different typed data in the same list.

```
AL = [360, Alan, "Erica", [6 7 8]]
```

defines a list with an integer constant, a symbol or variable name, a string and a list

Functions that are invoked in definitions are in a format that allows arguments to be passed in the form of lists. This follows the more usual practice where arguments are in parenthesis following the function name rather than the Lisp form. *e.g.* the definition that would in Lisp read as

```
ObjectA = (object basef size origin)
```

is entered into EdenLisp in the form:

```
ObjectA = object (basef, size, origin)
```

(Notice the commas between the arguments in normal Pascal or C style; round parentheses surrounding the arguments are also more usual than around everything.)

Conditional Expressions

We noted above that we can control the structure of expressions by the following form

```
If <logical> then <expr1> else <expr2>
```

where `<logical>` is a real expression interpreted as either true or false, and `<expr1>` and `<expr2>` are expressions (that may be further conditional expressions). Both `<expr1>` and `<expr2>` must evaluate to expressions of the same type as declared in the username and when the expressions are nested all the expressions that can supply meanings to the outermost condition must be of the same type.

Files

Input scripts of definitions are easiest to enter into EdenLisp by using text files since such files may be constructed using any text editor. (Text editors can co-exist with AutoCAD by means of a Windows environment such as X or MS-Windows, or using a Terminate, Stay Resident (TSR) package.) EdenLisp provides the AutoLisp function that is written

```
(Dload "userfile")
```

from the AutoCAD COMMAND prompt. The file has to be stored in the form

```
username.lsp
```

where in MSDOS the username must have a maximum of eight characters. EdenLisp assumes the suffix `".lsp"` is there. If the command is used within a file to insert another file then the EdenLisp will expect the EdenLisp form

```
Dload ("userfile")
```

with the parentheses around the argument.

5.32 Defining Abstract Objects

The structure described so far for EdenLisp is similar to many definitional notations (*e.g.* EDEN, DoNaLD and indeed PADL2). However when we come to defining objects we have a more abstract view, following the algebra based on the complex, frame, object types developed in chapter 4 (§4.22 and §4.23) where we introduced the terms used here. We describe the structure of complex, frame, object using the following fragment of EdenLisp code called **BOX** as illustration.

```
        ; BOX.LSP
1.      ; set up combinatorial structure
2.      LLstr : box
3.      Lstr  : AB BC CD DA EF FG GH HE AE BF CG DH
4.      AB = ["a","b"]
5.      BC = ["b","c"]
6.      CD = ["c","d"]
7.      DA = ["d","a"]
8.      EF = ["e","f"]
9.      FG = ["f","g"]
10.     GH = ["g","h"]
11.     HE = ["h","e"]
12.     AE = ["a","e"]
13.     BF = ["b","f"]
14.     CG = ["c","g"]
15.     DH = ["d","h"]
16.     box = [AB,BC,CD,DA,EF,FG,GH,HE,AE,BF,CG,DH]
17.     ; set up carrier structure
18.      point : a b c d e f g h
19.     a = [0,0,0]
20.     b = [1,0,0]
21.     c = [1,1,0]
22.     d = [0,1,0]
23.     e = [0,0,1]
24.     f = [1,0,1]
25.     g = [1,1,1]
26.     h = [0,1,1]
27.     ; create generic object frame
28.     frame : boxf basef
29.     boxf  = complex (box)
30.     basef = boxf
31.     frame : base0 base1
```

```
32.     ; create instantiation of object
33.     point : origin size
34.     origin = [0.0, 0.0, 0.0]
35.     size   = [100.0, 30.0, 20.0]
36.     baseO  = object (basef, size, origin)
37.     baseD  = Wireframe (baseO, "line")
```

We start with the most abstract form: the *complex* that is a list of subsets of labels. In the program it is a list of symbols representing the names for what could be sides of the box yet undefined. Line 16 is

```
BOX = [AB,BC,CD,DA,EF,FG,GH,HE,AE,BF,CG,DH]
```

Each of the terms `AB, BC, CD...` is defined as a pair of strings such as `AB =["a","b"]`. Elements pairs are not yet variable or identifiers, they simply indicate a combinatorial structure.

To derive a *frame* from a complex it is necessary to supply specific coordinates and scalar parameters corresponding to the abstract labels of the complex. So `boxf` needs to be supplied with definitions for the named variables `a b c d e f g h`. Rather as in PADL-2 it is convenient to have default values of unity for these values. Lines 19-26 define coordinate points of type `Lreal` (list of real) in terms of the unit box. Strictly one should specify the dimension of the space in which the complex is to be realised, but in EdenLisp the length of each coordinate list is coerced into that of the longest (by adding zeroes to the tail of the list). A warning is issued if the dimensions are not all the same, but it is allowed as it is sometimes convenient to enter a mixed 2D and 3D set in order to transform 2D into 3D.

The function

complex(box)

is the operator of the type "Operators for accepting complexes and lists of reals to realise a frame" described in §4.23. This operator accepts a complex and turns the strings (or quoted atoms) into labels and associates them with the pre-declared variables of that name. So line 29 returns the value

```
boxf   =  [(a b)(b c)(c d)(d a)
           (e f)(f g)(g h)(h e)
           (a e)(b f)(c g)(d h) ]
```

which is a frame. New frames can be constructed with the same default values by the equivalence operator, so line 30 creates the frame called `basef` that is identical with `boxf` but with a significant difference: the function **complex** yields the *names* of variables as its returned value, whereas the equivalence operator returns the

values of the variables associated with those names. The distinction between the two operators is a nice one but very useful for further manipulation.

Intantiation of a frame as an *object* is done by means of the function **object**, viz. line 36

```
base0  = object(basef, size, origin)
```

`object` acts as an operator on a frame plus two vectors. The first vector associates a scaling factor with each of the components of the corresponding variables. The second vector is a translation of the whole frame to a new local origin. Thus `base0` evaluates as follows (in AutoLisp format)

```
origin  =   (0.0 0.0 0.0)
size    =   (100.0 30.0 20.0)
base0   =   (object basef size origin) =
     ( ((0 0 0)(100.0 0 0)   (100.0 0 0)(100.0 30.0 0))
       ((100.0 30.0 0)(0 30.0 0)   (0 30.0 0)(0 0 0))
       ((0 0 20.0)(100.0 0 20.0)   (100.0 0 20.0)(100.0 30.0 20.0))
       ((100.0 30.0 20.0)(100.0 30.0 20.0)   (100.0 30.0 20.0)(0 0 20.0))
      )
```

Although this is now a set of vertices at a given position the object is only defined as a graph. We need operators on that graph to realise the object in the various ways we may wish to form the geometrical model: as a wire-frame or B-Rep, *etc*.

5.33 Operations on Sorts

The *type* FRAME is used for the complex and the frame depending upon the realisation of the graph structure that is employed. Complexes may start life as strings or symbols (Lisp allows either) but usually end up as lists of coordinates, *i.e.* `LLreal` so the operations on the algebra frequently change the type as the sorts are transformed from complex to the object.

Operations on complexes depend upon how the labels are specified. If they are strings then they can be concatenated with other strings to create new pairs with the same combinatorial structure. One method for doing that was first explored in CADNO by John Stidwell [Stidwell, 1989]. In EdenLisp we have a function ISOMORPH that acts as an operator on a complex with a string to create an identical complex in terms of its combinatorial structure but with each element of the input complex having the input string concatenated. So the function

```
isomorph("BOX", box)
```

concatenates the string "BOX" to each of the elements defined by AB BC CD... to yield pairs like AB =("BOXa" "BOXb"), etc. The advantage of this operator is that new abstractions with the same structure can be created but with different labels and hence with their own 'life' independent of their pedigree. Once the combinatorial structure is defined then the function **complex** transforms the complex to a frame in the way already described. So if we attach BOX in the way indicated to a new variable boxf1 then **complex** would yield the same structure with different labels:

```
boxf1  =  ( (BOXa BOXb)(BOXb BOXc)(BOXc BOXd)(BOXd BOXa)
            (BOXe BOXf)(BOXf BOXg)(BOXg BOXh)(BOXh BOXe)
            (BOXa BOXe)(BOXb BOXf)(BOXc BOXg)(BOXd BOXh) )
```

The ability of the complex to form new variables is very powerful. It allows us to localise information in a single instantiation or to pass information of a generic kind across a whole family of instantiations. The effect of redefining the basic form of BOX = (AB BC CD DA EF FG GH HE AE BF CG DH) would clearly change all derived objects. It would be the equivalent to redefining a primitive such a BLOck in PADL-2 during a session.

In fact with Lisp we can dispense with the need to have strings to form complexes. Using the quoted atom we can get exactly the same structure as with the string form excepting that we lose the ability to concatenate strings as before. The following amendment to the EdenLisp code for Boxes illustrates the difference.

```
      ; BOXES
1.    ; set up combinatorial structure
2.       LLreal : box
3.       Lreal  : AB BC CD DA EF FG GH HE AE BF CG DH
4.       AB  = ['a,'b], BC = ['b,'c], CD = ['c,'d], DA = ['d,'a]
5.       EF  = ['e,'f], FG = ['f,'g], GH = ['g,'h], HE = ['h,'e]
6.       AE  = ['a,'e], BF = ['b,'f], CG = ['c,'g], DH = ['d,'h]
7.       box = [AB,BC,CD,DA,EF,FG,GH,HE,AE,BF,CG,DH]
```

Since the complex is a graph structure we can have operators that perform edge creation for us. So in EdenLisp the following operators are provided

PEDGE: (poly-edge) takes a list and arranges it into pairs forming a consecutive set of edges like a polyline:

```
pedge(a,b,c,d) = ((a b) (b c) (c d))
```

CEDGE: (cycle of poly-edges) takes a list and arranges it into pairs forming a closed cycle joint first and last nodes:

```
cedge(a,b,c,d)  =  ((a b) (b c) (c d) (d a))
```

CGRAPH: (complete graph) takes a list of nodes and connects them in every possible way pairwise

```
cgraph(a,b,c,d) = ((a b) (a c) (a d) (b c) (b d) (c d))
```

Graphs may be combined by simply listing them. Other graph properties are easily added to these functions as required. For example paths and cycles could be extracted. (See implementation in chapter 6 for details of how to do that).

Frame functions are standard vector and matrix transforms on coordinates. The following functions are implemented in EdenLisp.

`Vsum (V1, V2)`	Vector sum
`Vdiff (V1, V2)`	Vector difference
`Vtrans (Rlist, Transl)`	Translate a vector Rlist through Transl
`Vscale (RRlist, Scalar)`	Scale an object by single scale factor
`Vshear (V1, V2)`	Scale vector by selective scaling
`SProd (V1, V2)`	Scalar Product
`InnerProd (vectA, vectB)`	Matrix inner product
`transpose (matrixA)`	Transpose of a matrix
`matrix-add (matA, matB)`	Matrix addition of 2 matrices of same size
`matrix-product(matA, matB)`	Product of 2 matrices of appropriate sizes
`Mshear (matA, vectS)`	Selective scaling of a matrix
`Vect-transform(matA, vectX)`	Isometry of a vector => vector
`rotV (vect, angl, axis)`	Rotation of a vector
`rotobj (lvect, angl, axis)`	Rotation of a set of coordinates

Frames can be used as the basis of sweeps, extrusions and similar 2D to 3D transforms, *e.g.* the function `extrude(cedge ([a","b","c","d"]))` takes the vertices closed cycle `["a","b","c","d"]` as argument and creates a new cycle of vertices `["ea","eb","ec","ed"]` and returns both with corresponding edge connections as an extrusion.

```
extrude(cedge(["a","b","c","d"]))
            = (("a" "b") ("b" "c") ("c" "d") ("d" "a")
               ("ea" "eb") ("eb" "ec") ("ec" "ed") ("ed" "ea")
               ("a" "ea") ("b" "eb") ("c" "ec") ("d" "ed"))
```

Apart from the transform function `object` described above, the main object type operators are to do with how a skeletal structure may be realised. Using AutoCAD's own commands one can realise a frame as line segments as in the left hand drawing in *fig. 5.2*, or it can be all circles to form a sphere. Or indeed it may be selective as in the cylinder formed on the right of *fig. 5.2*.

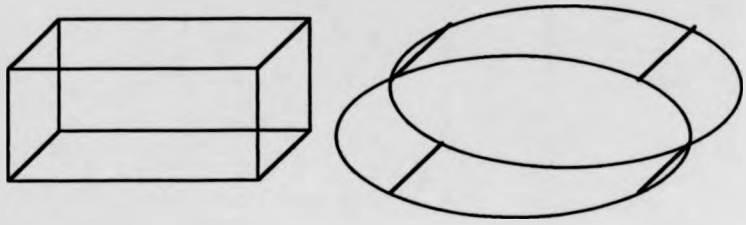


Fig 5.2 Two realisations from the same combinatorial structure

Realisation functions allow the user to specify the object in very different ways. The primary realisations are **`wireframe`** and **`Surface`** and CSG **`solid`**. **`Wireframe`** joins vertices by curves or straight lines as specified by the argument (*e.g.* `bard = Wireframe (barf, "pline"))`. The currently implemented realisations under wireframe include

```
"LINE"        . directed line segment
"SEGS"        . series of separate line segments
"PLINE"       . polyline
"SPLINE"      . splined curve through the vertices
"3DPLINE"     . 3D polyline
"ARC"         . circular arc
"POLYHEDRON"  . wireframe through the specified edges
"EXTRUSION"   . extrusion on a frame
```

`Surface` realisations follow a similar functional form, for example,

```
"3DFACE"      . 3D face using AutoCAD face command
"EXTRUSION"   . extrude to produce surfaces
```

Finally we can utilise AutoCAD's own CSG representation to display solids but with the added functionality of EdenLisp.

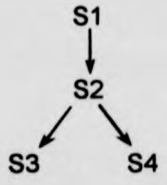
5.34 Windows

Objects made up of several components from the same root, such as assemblies of cuboids, need to be hierarchically structured so that parts can inherit properties that are above or generic whilst being able to have local properties independently of similar parts in the same family. The problem is analogous to having local variables in a procedure or function within a programming language. We may wish for example to have a box with the same labels for vertices `{a,b, .. h}` whether that box is on one part of the system or another, so saving having to remember whether labels have been used before. The labels have to be local. In EdenLisp that is achieved by means of Windows, rather in the manner of DoNaLD's `within{...}` declaration. Windows are structured as Unix directories, so that within a window, variables may be named regardless of their antecedents or descendants. Variables may be referenced from within a window structure if they not declared within the window (locally) but are declared in a parent window. If a parent variable is redeclared locally then that overrides the parent definitions whilst in that window but such variables only affect descendants, not antecedents when evaluation takes place.

A tree used to represent the window structure has nodes, each of which represents a particular context where new definitions may be constructed and that inherits all definitions on the path to the root node. Definitions created in a particular node can only be evaluated with information from that node or its antecedents on the path to the root. Redefinition or re-evaluation is not allowed from child contexts. An instantiation of a window taken from a remote node is allowed by making the imported variables and definitions local and independent of the source node.

An object is built up in nested windows with the most abstract information at the outermost window. An abstract shape is split into components: one set gives the underlying relationships used to delineate the shape, another identifies the way in which that shape will find representation. To illustrate the structure we create two objects with similar topology and geometry using windows `(S1, ... S4)` in a tree as follows.

```
(OpenWin 'S1)          ; Create and open Window S1
   LLstr : face$
   face$ = cedge ({"a", "b", "c", "d"})
                       ;complex definitions
   (OpenWin 'S2)
      point : a b c d O
      O = {0.0, 0.0, 0.0}
```



```
a = vsum (O, [-0.5, -0.5, 0.0])        ;frame definitions
b = vsum (a, [1.0, 0.0, 0.0])
c = vsum (b, [0.0, 1.0, 0.0])
d = vsum (a, [0.0, 1.0, 0.0])

    (OpenWin 'S3)                      ; Opens S3 off S2
      frame : faceF faceG faceO1       ; object definitions
      Real_vector : scale origin
      faceF  = complex (face$)
      Origin = [100.0, 100.0, 0.0]
      scale  = [20.0, 30.0, 0.0]
      faceG  = object(faceF,origin,scale)
      faceO1 = wireframe(faceG,"LINE")
    (CloseW 'S3)                       ; closes S3, moves to S2

    (OpenWin 'S4)                      ; opens S4, off S2
      frame : faceF faceG faceO2
      Real_vector : scale origin
      faceF  = complex (face$)
      Origin = [200.0, 100.0, 0.0]
      scale  = [35.0, 42.0, 0.0]
      faceG  = object(faceF,origin,scale)
      faceO2 = wireframe(faceG,"SPLINE")
    (CloseW 'S4)
  (CloseW 'S2)
(CloseW 'S1)
```

Nodes S1 and S2 provide the topology and abstract vertex positions for the two "face" objects that have different realisations in S3 and S4. Changing the abstract definitions in S1 or S2 affects S3 and S4 in the same way, but changes in S3 are independent of those in S4. So `faceO1` is a wireframe, `faceO2` is a spline.

The window structure of EdenLisp creates a definitive notation for Object representation with the following features.

a) Object representation is by means of trees of scripts that define geometrical and technological information at increasing levels of abstraction where the current state of an object is given by a single path in the tree. This path then represents the current state of interaction.

b) An evaluator of definitions accepts definitions from a script as input, and after checking, evaluates according to current information, transparently to the user and in any order.

c) The state of the display is defined according to the state of the object representation but may also show the current state of interaction and currently suspended states.

d) The graphical interface handles anything that the modelling system may wish to display.

5.4 Design Process: State and state change

State is represented in EdenLisp by the current set of definitions. Alternative states exist by redefinition, so are infinitely variable. In practice it is often desirable to constrain the state changes that can be made. Constraints are easiest to define via the `if` statement provided in EdenLisp. Using that statement in a definition makes the value of the definitive variable conditional upon another definition in a way that can be used to trigger re-definitions that constrain the user's ability to redefine (change state). At the weakest level a warning can be issued, at stronger levels preconceived changes may be triggered that counter, or follow, the effect of a change. For example a table lamp moved from its position may trigger an action to move the flex and plug to another, nearer socket. Alternatively the user may not be "allowed " to redefine a particular variable. (Clearly that is impossible to a super-user, who has ultimate power, but it can apply to an agent who is thereby kept from accessing a particular definition.)

A more subtle way of implementing changes due to constraint violation is the automatic rewriting of definitions. As explained above definitions that do that are called *actions* and can trigger wholesale rewriting. We investigate ways of doing such manipulation.

If we divide definitions into groups, or scripts, then each script may be operated upon more or less independently. Interactions with other scripts will be taken care of transparently. In principle each script could be changed by different agents. It would only be necessary to prevent interactions that would lead to illegal dependencies such as circular definitions, and that can be taken care of by suitable constraint management. What remains is whether those other agents can perhaps be the computer itself. For example it happens that certain scripts are very repetitious, involving large numbers of definitions that are broadly similar. Creating and managing those scripts is tedious and the computer can be asked to take over the task of changing the state of a script by writing or rewriting appropriate

definitions and implementing them as the new current script. That process of script manipulation then becomes a problem of definitive notations for defining definitions, a kind of meta-definitive notation. That meta-notation thus consists of state changing actions.

One way to create actions is pointed up by the experience of writing `shaft.DoNaLd`, namely by manipulation of scripts of definitions. There the scripts were managed by manipulating files of definitions, creating destination files by manipulation of text in source files. Using Lisp it is easier to manage such text manipulation. New definitions may be constructed as text or quoted lists and called into being as part of meta-definitions. In such an operation the text to be operated on could be the value of a definition of type string (or a list of strings). The action definition would have as its value a definition that was an amended version of the source text. Provided the new version was a legal EdenLisp definition and appropriate variables were correctly declared then the action would both evaluate the text changes and also send the amended text to the EdenLisp interpreter.

Another kind of action can create new variables and hand values to them. If a series of new variables is formed in sequence, each of integer value in increasing order, *e.g.*

```
p1, p2, p3, p4, p5, p6, ... ,
```

it is easy to see that an array can be constructed such as might be needed in forming a Cartesian graph. Actions would create the X and Y values that would be the plotted variables. If the string version of a formula in Y(X) is passed to an appropriate function then sets of pairs containing (Xi, Yi) can be computed. By that method individual points of the graph have separate variable name, rather than being accessed via a pointer into an array. There are times when it is useful to be able to access points in that way. For example the representations of points in different graphical format (bar, pie chart, scatter diagram) are well-known alternatives available in most computer packages. Using the actions described it is fairly trivial to use the points p1, p2, etc. to generate alternative formats for display purposes.

The power of actions is highly dangerous. It is possible to wipe out and re-write whole sections of code. However there is nothing except rule to prevent a prototype designer doing the same kind of damage to the product in either geometrical or physical form. The weapon of 'action' must be used circumspectly! And that matches the idea of constraint definitions described above. In combination

with "if" statements guards can be put on actions in the way that is described in connection with the Abstract Definitive Machine described earlier (section 3.42). If an action does do damage then some notion of state history is necessary. The simplest way to do that is to record the previous states by filing static positions. That is not a satisfactory way in the long term as it relies on the user doing a fair amount of storage, maybe most of it useless. This remains an area for further research.

6

EdenLisp Implementation

The design of a new language is best tested by its implementation. The author's language is no exception. In this chapter the detail design of EdenLisp is described. EdenLisp is a complete language with its own syntax and semantics to which user input has to conform. The compiler design, consisting of lexical analyser, parser and definitive evaluator is delineated, including the structure and function of the underlying symbol table. That symbol table is crucial to the ability of EdenLisp to permit operation in various environments that are suited to the different design requirements of the user. Related design objects can be ascribed tree structured related environments permitting global and local variables; the input environment is within AutoCAD, either directly or from a file, and the CAD environment is via AutoCAD's own CAD commands

6.1 Introduction to AutoLisp

AutoLisp®, the programming interface to AutoCAD, is used to implement EdenLisp. The functional programming style of Lisp lends itself well to the prototyping of definitive notations. AutoLisp is interpretative: input Lisp instructions are immediately evaluated and the result actioned. Its syntax and structure are as Lisp, but it includes access to all commands that can be issued at the AutoCAD Command prompt. Files of AutoLisp code are loaded by typing at the Command prompt

```
(load "filename")
```

where *"filename.lsp"* is an ASCII text file of AutoLisp statements. The suffix *".lsp"* is necessary when writing the file, but is understood in the *load* statement. The parentheses are a necessary part of Lisp and rather awkward since nested statements may require large numbers of brackets (prompting some to say

that LISP is an acronym, not for LISt Processing, but Lots of Irritating Silly Parentheses!)

The AutoLisp environment makes editing and debugging AutoLisp programs difficult as AutoCAD has to be running to interpret the code. The solution adopted was to use a TSR (terminate, stay resident) editor such as SideKick® or PCTools®. The latter has the advantage of allowing multiple files to be edited and both can be called whilst AutoCAD is running in text mode. With recent developments in Windows environments these problems will disappear.

A brief introduction to AutoLisp is useful as background to EdenLisp. Since AutoLisp has the same syntax as Lisp the reader is referred to texts on Lisp for more details, [*e.g.* Winston & Horn, 1989, Jones, Maynard & Stewart, 1990]. Lisp is one of the oldest computer programming languages and in some of its logic it actually antedates the modern digital computer. Its power arises from its ability to attach information to symbols. It only has one data-type, namely: ***symbolic expression***, divided into ***atoms*** (numbers and symbols) and ***lists*** (of atoms or lists).

The following examples show how the Lisp interpreter evaluates symbolic expressions and prints the result. `$` is the Lisp prompt.

```
$ 4                  ; 4 is a numeric atom that evaluates to itself.
4
$ t                  ; t and nil are special atoms that evaluate to
T                    ;   themselves. nil is false; everything else is
true.
$ (+ 4 (* 7 8))      ; Inner arguments 7 and 8 are evaluated, the product
60                   ;   found and the result passed to the + function.
```

Naming is carried out by operators. `Setq` is an assignment operator that associates a variable with a value and returns that value. Because it makes a permanent change in a variable `setq` is one of a class of operators called ***mutators*** that are not part of a pure functional language. To associate the value `6` with the name `a` we write:

```
$ (setq a 6)
6
```

The addition of a single quote before a symbol forces the interpreter to take the name and not its value. The assignment

```
$ (setq b 'a)
a
```

is interpreted as "set the value of *b* to the name *a*", hence it returns *a* and not *6*. `Setq` actually means *set quote*, hence "set the name b to the name *a*". That property is useful in being able to manipulate names, create macros and even create lisp within lisp. EdenLisp uses that property to advantage.

The task of keeping track of the assignations is carried out in the *environment*. Different assignments may be made if environments can be make local. AutoLisp allows local variables within functions.

Lisp has three basic list processing operators.

```
$ (cons 4 nil)     ; cons converts atoms to lists
(4)                ;   or adds an atom to a list
$ (car '(a b c))   ; car returns the head of any list.
a                  ; The head may be an atom as here, or a list
$ (cdr '(a b c))   ; cdr returns the tail, the remainder of a list after
(b c)              ;   removing the head. Result is a list (may be empty)
```

From these operators further lisp functions may be built up by the operator *defun*. Functions are made available for use in any program by simply calling it by its name with its arguments, *e.g.* the following defines the function *member*. Line numbers are added for reference; they are not part of Lisp.

```
(defun member (item lista)                          1.
(cond                                               2.
 ((null lista) nil)                                 3.
 ((equal item (car lista)) lista)                   4.
 (T (member item (cdr lista)))                      5.
))                                                  6.
```

The function tests if *item* is in *lista*. So

```
$ (member 'g '(b c d e f g h i))
(g h i)
```

takes as input the symbol *g* and tests if it is in the list *(b c d e f g h i)*. If *g* is found then it and the remainder of the list is returned. Since the answer is not *nil* it is also interpreted as *TRUE*.

The function uses *recursion*. The *cond* statement is similar to the *case* statement in procedural languages. There are two conditional cases and an "else" case.

- if *lista* is empty then exit the function returning *nil*

- if *item* is the same as the head of *lista* then exit returning *lista*
- if none of the other cases apply the "else" statement denoted by T causes the function to be invoked again but with *lista* without its head.

When *member* is first called the two cases in lines 3 and 4 fail so the **r** clause causes the function to be called again as *(member 'g '(c d e f g h i))*, i.e. without the first symbol *b*. That call also fails, so *(member 'g '(d e f g h i))* is invoked... and so on until *(member 'g '(g h i))* is invoked when line 4 will succeed, returning *lista = (g h i)*. At that point the function has been called 6 levels deep so has to climb out 6 times before delivering the result. Results of intermediate levels are stored on the stack.

The advantage of recursion is the brevity of the code required, making prototyping very rapid, if rather dense to understand at first. The disadvantage is the stack growth. On AutoLisp the stack is 146 levels deep, beyond which the interpreter overloads. The code must then be written to make it "tail-recursive" (i.e. to make the recursion carry its own previous result so that the depth of recursion is reduced to the levels an iterative method would use).
AutoLisp has built-in functions that are common to most versions of Lisp to save defining them (*e.g. member* is a built-in function). A list of AutoLisp functions accessible to users of EdenLisp is included in Appendix 1.

The simplicity of an untyped language like Lisp is the ability to create abstractions on data that can capture primitive ideas in functions that in turn become building blocks for higher order structures. As Abelson and Sussman put it in their seminal book on Structure and Interpretation of Computer Programs:

> "As we confront increasingly complex problems, we find that Lisp, or indeed any fixed programming language, is not sufficient for our needs. We must constantly turn to new languages in order to express ideas more effectively. Establishing new languages is a powerful strategy for controlling complexity in Engineering design." *[Abelson & Sussman. 1985]*

6.2 The EdenLisp Compiler

EdenLisp consists of AutoLisp functions grouped into files as follows

EDENLEX.LSP	Lexical Analyser
EDENPARS.LSP	Parser
EDENENG.LSP	Main Engine: input functions, interpreter and general control

EDENTYPE.LSP	Type checker
EDENENV.LSP	Environment control: Symbol table, object/frame management
EDENEVAL.LSP	Recursive Tree Evaluation of definitions
EDENUTIL.LSP	Utilities and error trapping

Geometrical and topological functions, including draw functions and operators are in files *EDENGEOM.LSP* and *EDENTOP.LSP*. Actions are in *EDENACTN.LSP*.

Fig 6.1 shows the organisation of the EdenLisp "compiler" in a block diagram. The lexical analyser scans the stream of EdenLisp source code, and separates it into tokens. The tokens are lisp *dotted-pairs* such as `(a . b)`. The latter have the property of being stored more compactly in memory than lists such as `(a b)`; furthermore, `(cdr '(a . b))` returns the atom `b` rather than list `(b)`. The head of the dotted pair identifies the nature of the token and its tail the actual keyword such as `IF`, `THEN`, or predicate symbol. The token stream is the input to the next module, the parser.

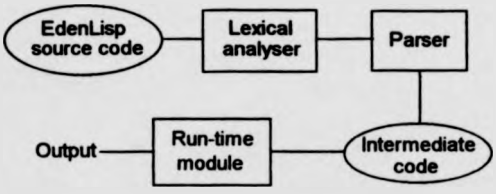


Fig.6.1 Organisation of the "compiler" stage of EdenLisp

The parser groups tokens together in accordance with their syntax into a parse tree. For instance an EdenLisp definition is a syntactic structure consisting of tokens arranged in the form: `<variable identifier, assignment operator, formula>`. A formula is any collection of symbols that are conventionally used to describe a relation. Input is expected using infix notation such as `(a + b)` rather than the prefix notation `(+ a b)` used in Lisp. In the parser the parse tree is traversed and the structure altered into the Lisp format. The parser also identifies dependent variables in the formula and the type of EdenLisp statement being input. The output lists the EdenLisp statement type, independent and dependent variables and the predicate in AutoLisp code. The lexical analyser and parser are discussed in the next two sections.

6.21 The Lexical Analyser

The lexical analyser "EDENLEX.LSP" accepts a string as input and reads it from left to right. Each character is read off the head of the string and the string replaced by its tail. The process uses a `while` statement rather than recursion in order that a finite state machine (FSM) can be used to implement the lex stage.

Each character is identified by the lexical scanner into one of the character groups shown in *table 6.1*. Character groups are used as states in the FSM. For example, if the input string is *"phi = sin(a)"*, then the following state transitions occur within the FSM (see *fig. 6.2*). Starting in state NEW, character *p* causes the FSM to move to the state ID, and *p* is pushed onto a stack.

RULE	Token Name
nothing	*nil*
" ", "\t"	'SPC
string char in [! ( ) , ? { } &]	'PUNC
"\\", "[", "]"	'PUNC
string char in [+ - * / ^]	'DEFOP
string char in [< > | ~]	'RELOP
"`"	'STR
string char in [a..z, A..Z, _ $ % @]	'ID
":"	'COLON
"="	'EQUAL
";"	'COMENT
string char in [0..9]	'INT
"."	'DOT
"'"	'SQUOTE
"\042"	'STR
any other character	'PUNC

***Table 6.1**. Characters are recognised into tokens named in the second column*

The next character, depending on its character group, moves the FSM to a new state or keeps it in the current state. In our example, *h* is received and added to the stack, the FSM remaining in state ID. The same is done with the *i* character. Receipt of the *spc* (space) character ends the ID state: the current stack is emitted and the FSM returns to NEW. The *spc* character, as all white space, is ignored and the machine simply scans for the next character. This allows the input to be "pretty printed" by the user. Similar transitions are shown in *Fig 6.2* for the remainder of the definition. A definitive statement is ended by `return` (chr 13)

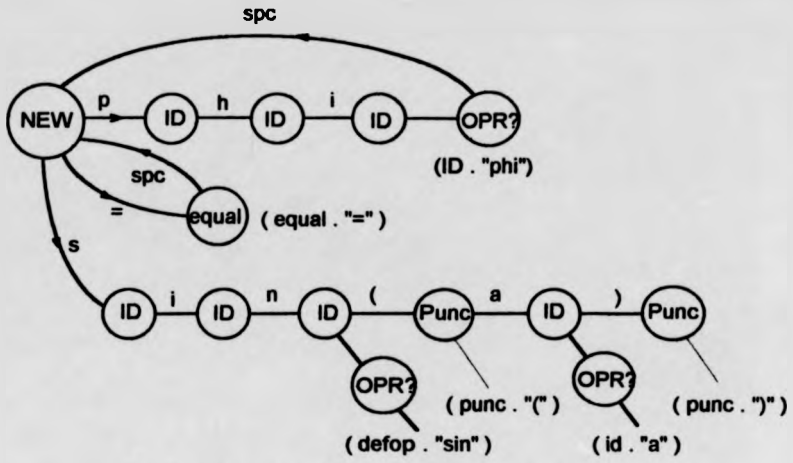


Fig. 6.2 State transitions for input stream `phi=sin(a)`

6.22 The Parser

The parser *"EDENPARS.LSP"* accepts a stream of tokens (dotted pairs) generated by the lexical analyser. The stream is of finite length and is first tested to be one of the following valid EdenLisp inputs.

1. a type definition of the form `type : id (id..)`
2. a definitive statement
   - with non-graphical output, of form `id = const|id|formula`
   - with graphical output, of form
     `id = draw_function|draw_function_formula`
3. Special definitive form: specified dependent variables are to be held at the values that are current on definition. E.g.
   `a : b c : b+c/d`
   where `b=8.0, c=7.5,` implies that `a = 8.0 + 7.5/d` is the definition to be evaluated thereafter, whatever subsequently happens to `b` and `c`.

Forms 2 and 3 are separated into left and right sides of the assignment symbol. Provided it is syntactically correct, the right side is converted into a parse tree using grammar rules; otherwise an appropriate error message is generated. The parser uses a *recursive descent parser* (c.f. Tanimoto, 1990) to change the input stream infix order into the corresponding prefix notation. The twelve non-terminals: *program, statement, expression, relational_expr, term, factor, opr1, opr2, int, real, id,* and *function* use production rules that are shown in Appendix 1.

An input submitted to these rules is treated recursively. The output, for example from `a+b+sin(c)+d` returns the binary form `(+ a (+ b (+ (sin c) d)))`, rather than the more compact form `(+ a b (sin c) d)`. Although both are Lisp compatible the binary is easier to generate. The recursive form is not the most efficient parsing method but has the advantage of compact coding, hence easier to debug.

The parser also allows input of standard AutoLisp. If an opening `"("` is detected in the definitive formula then the formula is handled directly by the lisp interpreter in the same way as Basic allows Peek and Poke. Errors are then handled by AutoLisp. Another AutoLisp form allowed is an opening `"!"` such as `"!a"`, which AutoLisp interprets as `(print a)`, useful for debugging and interrogation of the variables.

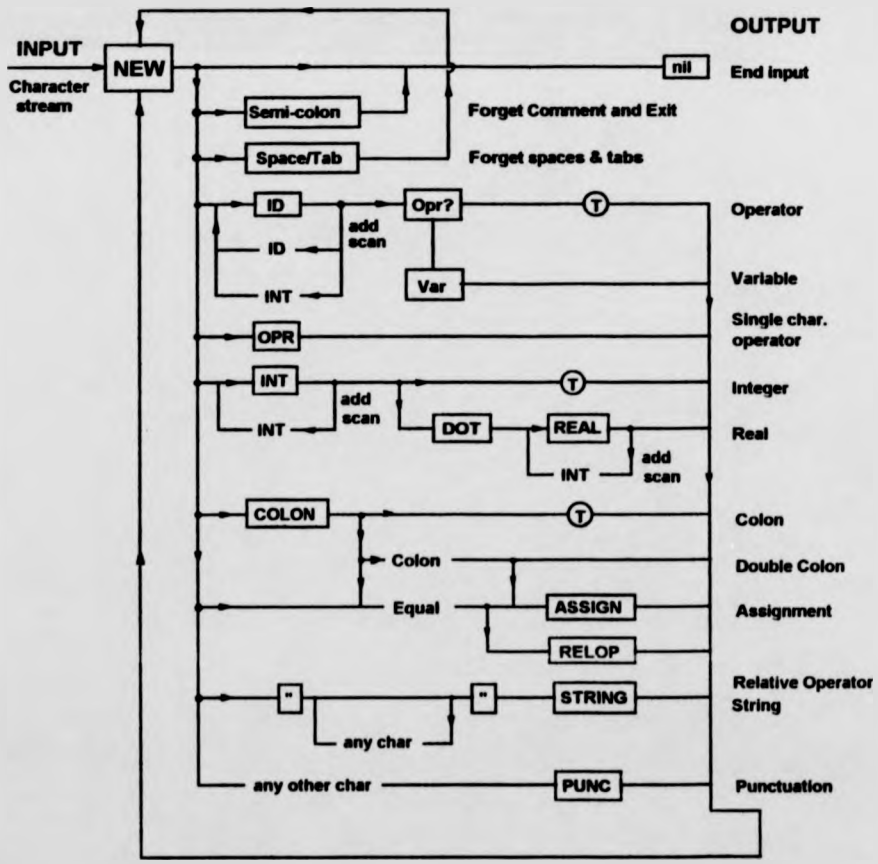


Fig. 6.3 Finite State Machine for Lexical Analysis of String Input Stream

Finally the parser makes one or two other translations in order to format the grammar more conveniently. This is best illustrated by an example.

```
$ (parse (lex "a = b+(c - d/e)"))
(DEFN "a" (b c d e) ("(+ b (- c (/ d e)))") )
```

At the Lisp prompt the command *(parse (lex ..))* causes the parsing of the lexical tokens obtained from the definition, returning the form shown in the next line. The parsed form has four or five components

1. the kind of EdenLisp statement (type definition, definitive statement, restricted definition)
2. the actual variable or type depending on the kind of statement as a string
3. a list of dependent variables
4. the formula, definition or list of variables to be typecast, in string form
5. if restricted a list of restricted variables

6.3 Definitive Interpreter and Symbol Table

6.31 Identification of Statements

The intermediate code from the parser input is passed to the EdenLisp "engine", *EDENENG.LSP*. The engine identifies the type of statement using information at the head of the code and then deals with each as appropriate. As observed above, a statement must be a Lisp statement, a type declaration, a definition, or a restricted definition. If it is not an input error is returned and the interpreter ignores the whole statement.

1. **LISP** statement

Lisp statements must begin with an opening parenthesis "(" so that is always taken as the signal that the statement following is in AutoLisp. The statement is passed through to the AutoLisp interpreter without being actioned or stored by EdenLisp. In that respect such statements are akin to the commands in PADL-2. Care is required with the use of such statements. Rather like using assembler from within a conventional language unexpected things may happen because EdenLisp does not know about any changes made by Lisp statements. Thus subsequent EdenLisp instructions will be interpreted in the new environment.

2. **DECL** = Type declaration, input is of the form

< DECL, name of type, string containing a list of variables to be declared >

The parser has already checked the name of the type is legal, so the string is now converted into a list of identifiers. The existence of an identifier is checked by scanning the symbol table. If a record is found in the symbol table the interpreter refuses that declaration and returns a warning to the user. If the declaration is valid the engine passes it to the environment control for creating a new record to hold information about the variable. (The environment and symbol table are dealt with below.)

3. ID is legal

Regardless of form of the statement, if it is not one of the first two kinds then it is some kind of definition. That being so it must be of the form `ID = statement`. The interpreter can therefore check at this stage to see if ID had been declared. If it has not, then an error is returned and the interpreter ignores that statement.

4. **DEFN**. Definition or redefinition of form `ID = statement`.

The interpreter first decides what kind of statement is being defined and actions each slightly differently.

a) If ID is a constant or a string that is accepted and sent on to the evaluator

b) ID is a formula or a draw function. Dependent variables are extracted from the parsed input string and checked to see if they have been declared. If they are declared then the interpretation proceeds; otherwise an error is returned and the interpreter ignores that statement.

5. **RDEFN**. Definition or redefinition of the form `ID : Dvars : statement`

This indicates a restriction is placed on certain of the dependent variables (`Dvars`) in the definition. The purpose is that those `Dvars` between the colons are to be evaluated immediately and the current values of those variables entered in the definition as constants. For example the RDEFN defined as follows

```
a : b c : b - c * d
```

would be evaluated by setting `b` and `c` to their current values and then passing the resulting formula on to the evaluator. If for example `b=8` and `c=b-4` then `a` would be defined as

```
a = 8 - 4 * d
```

No subsequent change in `b` or `c` would affect the definition of a since neither `b` nor `c` appears in the formula.

If the input takes the form of a definition then the interpreter has a number of functions to carry out

1. The ID may occur on both sides of the definition. This is a circular definition and would cause an infinite loop. Reject.
2. The definition is a constant. Test the type of the constant against that of the ID. If OK then proceed at step 5.
3. The definition is a formula. Check dependent variables and those variables that depend upon ID and list them, checking for hidden circular definitions. Go to 5.
4. The definition is to replace an existing one. Store the previous symbol entry in case an error is generated during evaluation. Go to 5.
5. Submit the new formula to the evaluator

Circular definitions such as `a=a+1`, unlike the equivalent procedural statement `a:=a+1` that merely adds one to the variable, would cause the evaluator to try to update variable `a` over and over, forever. Circular definitions may be caused unwittingly by a series of related definitions where a variable ends up after several definitions being defined in terms of itself. Thus the interpreter has to test all dependent variables for circularity before allowing evaluation to proceed.

6.32 Type Checking

Type is tested by functions in *EDENTYPE.LSP*. This was a formidable task. EdenLisp, unlike Lisp itself, is strongly typed. Each variable must have a declared type and operators may be defined with meanings appropriate to particular types. This has numerous advantages. For example one could unambiguously define the following on the operator +

```
int+int=int | int+real=real | real+real=real
point+point=vector_sum or point+point=line
line+line=polyline
```

The method adopted was to structure type checking by means of "manifest types" [Abelson & Sussman, 1985, 2.3.2] whereby a data object has a type that can be recognised and tested. That means that all variables must have their type stored alongside the identifier. The symbol table record looks after that so the type checker can replace the variables in a formula by their pre-declared types and then evaluate the type that results from rules such as those quoted above for the "+" operator. The type evaluation can be as complex as the evaluation of the formulae themselves, since one has effectively the same number of variables and operators as the other. That makes the task of evaluation up to about twice as difficult in the

worst case although certain simplifying operations such as coercion may be possible. Coercion is included in EdenLisp: `INT` may be coerced to `REAL` if appropriate (*e.g.* `3+6.9` gets coerced to `3.0+6.9`). Lists of integers are automatically coerced to `LREAL`.

In the current implementation the only binary operators are simple arithmetic on numbers. Most operations are carried out using functions on the given types. Lists of these functions and the types of variables passed as arguments are listed in Appendix 1 together with the types returned by those functions. AutoLisp functions that are compatible the EdenLisp types are included. Those correspond to common numeric operations (such as square, root, log and trigonometric functions) and certain list operations on which EdenLisp puts a greater structure than just *list*. Realisation functions currently return LLreal with display done by side effect.

List properties are used to advantage in checking formulae that have many operations of the same type. A formula such as

```
thetam = log(4*K*R+Omax) / cos(Kt*E-th)
```

has all real operations and the result is real. Thus all that is needed is to find the types of all variables and check they are all the same. The same technique is also used where there are different types in the input but some are repeated.

The main difficulty encountered in type checking was dealing with one of Lisp's most powerful devices the quote function. Quoted atoms or quoted lists are not evaluated under any operation so `('a+'b)` has no meaning, whilst the function `(openWin 'Swin)` is interpreted as "open a window called `Swin`". In the latter case `Swin` does not have a value unless one is assigned separately. Since variables in a quoted list may well be declared and have a type it is vital that the quoted list is excluded from the type check and replaced by `QATOM` or `QLIST`.

6.33 Symbol Records

EdenLisp symbols are held in a *symbol table* that has the AutoLisp variable name `*sym*`. Symbols are variables with types defined by the user through the type definition statement. On receipt of a valid declaration statement a *constructor* function creates an empty symbol entry that is a **record** containing first the name or identifier of a variable and then a series of at least one list of attributes. Each set of attributes appertains to the properties of a variable of that name in a particular window and lists of attributes are stacked so that the current attributes are those in

the current environment (see below for environment). Attributes are **fields** in the record as follows.

IDval:	The current value of the *ID*. If undefined then symbol `e` is used.
IDtype:	The declared type
Def:	The definition itself
DefType:	The definition type (*const, formula, drawfunction,* etc.)
DepVar:	Name(s) of dependent variable(s) in the definition
DepOnIt:	Name(s) of variable(s) that depend on this *IDval*
handle:	This contains a value that is generated by an AutoCAD command if *DefType* is a Draw Function. Its value is the address of the entity information in the AutoCAD database.

For the input *"int : a"* the constructor will create the record

(a (@ int nil nil nil nil nil))

This record is appended to the symbol table. Values of individual fields of a record of a single variable can be inspected by means of *selector* functions. (All functions of this kind commence with an underscore character.) All these:

`_idVal, _idtyp, _Def, _Dtype, _Dvar, _DOnIt, _Handl`

take the `id` name as argument. One has to be in the correct environment to access that information as the current window is assumed.

To change the values of any field in the record we use a ***mutator*** function, so-called to emphasise that an assignment (a permanent change) is being made by means of that function. These functions all have an exclamation character to emphasise they are mutators. The function calls need the `id` name, the new value and the name of the window (`win`) that is the home of the variable.

```
IdVal! (id idVal win)
Idtyp! (id idtyp win)
Def!   (id Def win)
Dtype! (id DType win)
Dvar!  (id DVar win)
DonIt! (id Donit win)
handl! (id handle win)
```

6.34 Evaluation

Evaluation of definitions is carried out by *EDENEVAL.LSP* functions and calling the mutator functions. The first task is to check whether the definition of a variable ID over-writes a previous one. If it does, then the symbol table needs to be amended

to cancel the previous definition. The previous form of the record is put in a temporary store in case it is required to be put back if an error is encountered during evaluation. All records that appertain to the previous definition need amending. The previous dependent variable list is used to find those records and the name ID is deleted from each of the "DepOnIt" fields of the dependent variables. (The significance of the "DepOnIt" field is that the new variable must be changed by any subsequent change in a dependent variable.) We now proceed with the new definition. First the records of the new dependent variables are fetched, ID added to their "DepOnIt" field and then the new definition is passed to the evaluator.

The evaluation proceeds downward and upwards: downward to find values of dependent variables and upward to amend variables that depend upon the value of the new definition. Moving down the dependency graph simply means finding values of dependent variables by using the Lisp interpreter on the formulae in their definition. However the upward traversal may be complex as the new value of ID may mean that those variables that depend on it now have values, they previously being undefined. The only way to check that is to evaluate the variables that depend upon ID on the whole path to the root. To do that will mean invoking downward evaluation for each variable on the upward path. Recursion is used extensively for this process.

6.4 Environment

6.41 Window Environment

The evaluation strategy described satisfies the requirement for a single set of definitions for an object. However in our algebra we wish to group different objects that have common properties. That in turn requires that we group symbols in places that are according to the object to which it appertains. In EdenLisp we follow a similar arrangement to that suggested by [Abelson & Sussman, 1985, §3.2] where these places are maintained in structures called *environments*. An environment is a sequence of *windows*. Each window is a table in which the first entry is the name of the window, the second entry is the name of the parent window (or nil if it is the root window), and further entries are *records* of variables containing their names and their *bindings* that associate variables with values & definitions. (A window may contain at most one binding for any variable; if there are more than one the variable is coerced to the last definition of that variable to be input.) The *value* of a variable with respect to an environment is the value of the

variable in the first window in the environment that contains a binding for that variable. If no window in the environment specifies a binding then that variable is ***unbound*** in that environment. In practical terms that means if there is no record in the symbol table the variable is unbound. A variable may be declared and not be given an explicit value. The value returned in that case is the special cipher **e**.

To explain the windowing environment we use the following example of EdenLisp.

```
(openW 'S1)
int : a b c d
a = b + c
b = 4
d = a
     (OpenW 'S2)
     c = 3
     b = c
          (OpenW 'S3)
          a = 9
          c = 10
          (closeW)
          (OpenW 'S4)
          a = 8
          b = 14
          (closeW)
     (closeW)
(closeW)
```

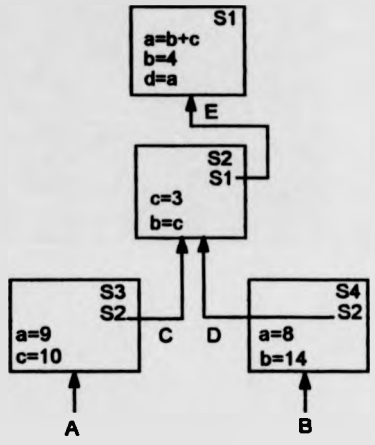


Fig.6.4 A Simple Environment Structure

Fig 6.4 shows the environment structure consisting of four windows S1 to S4. In the figure A, B, C, D and E are pointers to environments with C and D pointing to the same one. Environment paths are indicated by having the name of the window followed by its parent. Variables **c** and **a** are bound in the S3 window, while **b** and **d** are bound in windows S2 and S1. The value of **c** in environment D is 3. The value of **c** in environment B is also 3. This is because **c** is not bound in B so a binding is sought in the enclosing environment D and we find a binding in the parent window S2. On the other hand the value of **c** in environment A is 10 because the first window binds **c** to 10.

There are a number of problems that make the environment structure difficult to implement in AutoLisp. First, variables accessed outside functions are necessarily global. If we enter the lisp statement

```
(setq a 8)
```

then the variable **a** is set to 8 and cannot be changed locally in the manner described. The way that this was addressed was to have copies of the local value stored in the record appertaining to the variable and the record stored in the particular window where it is declared. When a window is opened, the function scans all the existing variables in the window and sets them to the values in the appropriate record. A variable can then be global for the purposes of the evaluator but its current value is always stored back in the record whenever it changes.
The second problem arises in relation to variables such as **a** being changed in a child window. How do we make sure that the value of **a** is different in environments A and B? In the implementation described above the record of **a** will be changed by the redefinition of `a=9` in S3 or the redefinition `a=8` in S4. That means that separate definitions are required for S3 and S4. Worse, when in S2 there is yet another definition for **a**. So where do we store these different records and make sure they are accessible?

In the above scenario the variables are only declared in the root window S1 so all the changes are stored in S1 and none in the other windows. So what solutions are possible?

1. LOCAL: If a redefinition is made of a variable from an antecedent window in a child window, then create a new record to be kept in that child window. Such a *local variable* would have to be declared locally and would be separate from the one of that name in the antecedent window. The difficulty of that is seen in window S3. Here c=10 is defined, but in S2 b=c. Thus the evaluation of c=10 will necessarily update b to have the new value of b. In S2 b=3 whereas in A b=10. The question then arises as to where to record the value of b in S3. It would not have been explicitly declared as local in S3 since there is no direct redefinition of b in S3. If the record in S2 is updated there is no way of re-establishing the value of b when returning to S2 except by reconstructing the script appropriate to that environment and evaluating its definitions all over again. We can do that by *undoing* new definitions in the current window before moving to any antecedent and re-evaluating using the definitions found there.

2. An ENVIRONMENT based system would have the environment path as the only current one that can be changed. When a new window is opened all antecedent variables have a new attribute set added to their current record, a copy of the existing attribute set. For example the record

```
(a (3 int 3 const nil nil nil))
```

would become the record

```
(a (3 int 3 const nil nil nil) (3 int 3 const nil nil nil))
```

when a new child window is created and entered. The current record is now regarded as the one immediately following the id. This process is called 'pushing' the stack of attributes. That process means that all records in a given environment will be updated to be appropriate to that environment and any subsequent changes will be correctly entered into the most recent of the records regardless of where the record is in the environment. Thus in environment A the values of (a,b,c,d) will be (9,10,10,9) while in environment B they will be (8,14,3,3). On exit to a parent window the most recent record attributes may be 'popped' off and the records are then in the correct state for that window. If the window already exists when the use transfers to it, then the definition in that window will need to be re-evaluated to update the records on entry to that window. An alternative to that would be to carry the window title along with the attribute stack so that popping is not done but the attribute set appropriately to a given environment can be detected.

The multiple attribute system is implemented in the current EdenLisp.

The operations on environments that are required are implemented in EDENENV.LSP in terms of operations on windows and bindings. These functions are set up as far as possible as constructors, selectors and mutators. *Constructors* set up data in the correct format, selectors allow inspection of data and mutators change the data. Examples of each kind are as follows.

Constructors

```
(Mk-Win W-name)
(Mk-Rec id idval idtyp Def Dtype Dvar DonIt Handle)
```

Mk-Win constructs a new window as the child of existing environment and Mk-Rec constructs the empty record described above *e.g. (a (@ int nil nil nil nil nil)).*

Selector functions interrogate the symbol table by window, record or field as follows. Some names have been introduced without the underscore because they are familiar names for those kinds of operations.

```
(_win W-name)              ; get window from *SYM* table
(_pwin W-name)             ; get full parent window of win
(_ChWin W-name env)        ; get names of child windows of win from *sym* table
(_ids Win)                 ; list all entries of variables in current window
```

```
(dir W-name)                ; list window name, parent, all variables and child windows
(path Win)                  ; list the path from win to the root
(_rec id W-name)            ; get complete record of id from *sym*
(_fld fldNo id W-name)      ; get a particular field from a record of id
```

Mutators are functions that change an environment, window, record or field

a) Environment Mutators

```
(MkEnv!)                    ; sets up startup global Symbol table *sym* and *win*
(ClrEnv!)
(setV! Reclst)              ; sets vars in each rec to the value in a second position
(reval! parent)             ; re-evaluate on closing a window
```

b) Window Mutators

```
(OpenW W-name)              ; Open a window called W-name if it does not already. Insert
                              the new window in *sym*. If the window exists then set
                              variables to local values and set the constant *win* to the
                              new window.
(CloseW )                   ; Close current window and move to parent. Reset variables to
                              local values of parent and set *win* to new window = parent
(Ins-W! W-name win)         ; Include a new copy of existing window W-name in current
                              win
```

c) Record Mutators

```
(New-R! rec win)            ; Insert a complete new record in win
(ins-R! id rec win)         ; Change complete record in win. That involves getting the
                              existing window and record for changing, replacing the old
                              record with rec, putting rec back in win and then replacing
                              win in *sym*
(PushR! id)                 ; Make a second copy of the record (other then the id) before
                              the current record such that rec = (id (copy rec) (previous
                              rec)). Amend the previous record by appending the current
                              window name (*win*) to indicate the position of current rec
                              in its environment
(PopR! id childW)           ; Remove the most recent attribute list in the id record.
                              (undoes the effect of PushR!)
(ins-F! newfld fldNo id env) ; Replace the field fldNo in record id in current window
```

6.42 Programming environment

The user environment for EdenLisp is within AutoCAD itself. The EdenLisp interpreter is loaded from the AutoCAD command prompt by means of a Lisp command file. The user simply types

```
(load  "Edenu")
```

and it gets loaded. The parentheses are essential as that signals an AutoLisp command. The AutoCAD command `load` without parentheses has quite a different meaning.

(The above assumes the relevant files are in C:\edenlsp. If not the full path name must be written between the double quotes. AutoCAD uses the backslash to signal that a special character follows so it is necessary to type two backslashes within quotes *viz.*

```
(load  "c:\\edenlsp\\Edenu")
```

AutoCAD also accepts a single forward slash in such circumstances, as in Unix; *i.e.*

```
(load  "c:/edenlsp/Edenu")
```

is equally acceptable.)

Once EdenLisp is loaded the interpreter is run by the AutoLisp command

```
(eden)
```

after which the user is presented with the EdenLisp prompt `:>` Only valid EdenLisp or AutoLisp statements are allowed at the prompt. Previously written EdenLisp scripts must be text files (ASCII files in MSDOS) and may be loaded either from the Command or the EdenLisp prompts by the `DLOAD` function described earlier.

Exiting EdenLisp simply requires the ENTER key to be struck.

The method described is the formal way to load EdenLisp. AutoCAD can be customised in many ways and the pull down menus in Version 10, 11 and 12 can be used to make it easier to load not only the EdenLisp interpreter but also any EdenLisp programs.

Since EdenLisp consists only of AutoLisp functions it is possible for it to coexist with any other AutoLisp functions, although care is required to prevent functions of the same name being loaded, with perhaps very peculiar consequences!

The symbol table is set up at the first invocation of `(eden)`. If it is necessary to clear the symbol table, as one might if a new set of objects or a new program is to be entered then `(ClrEnv!)` `(MkEnv!)` are the functions used to clear the Symbol table and to create a new one. These functions do not clear the screen so it may be

necessary to invoke AutoCAD commands to do that. `Undo back` will undo all commands made in AutoCAD up to the previous IO command. EdenLisp itself, as with all AutoLisp commands, can only be unloaded by explicitly setting all functions to nil. That fearsome task is best done simply by exiting AutoCAD and starting a new drawing.

The program structure for a script is easy if there is only a single (global) environment. Variables are declared and definitions made on those variables. If a windowed environment is desired then the `(openW WinName)` function sets up the window in the symbol table and sets the global variable `*win*` to be the current window. All subsequent declarations will be within that window and a new variable with the same name as one in a parent window is allowed and coerces the variable to have the child definition whilst in that environment. Windows opened must be closed by `(CloseW)` to cause the environment to revert to its parent window. If the `openW` command is explicitly used to open a parent, the child window is automatically closed. `(CloseW)` does not need an argument as the current one is implied.

6.43 CAD Environment

The CAD display environment is AutoCAD. Objects are defined, manipulated and displayed by means of AutoLisp functions intended to be accessed from within EdenLisp. These functions are arranged into the files `EDENTOP.LSP`, `EDENGEOM.LSP`, `EDENDISP.LSP`. The first of these, `EDENTOP.LSP`, contains graph theory functions such as `PEDGE`, `CEDGE`, `OBJECT`. These functions manipulate labels as lists of strings or quoted names. `ISOMORPH` is an interesting function that creates a new list of strings identical in arrangement to the input but distinguished by concatenating the second argument (a string) to each of the component strings. As the function name implies the output has the same shape (or graph) as the input. This provides us with the means of creating instances of the same topology whilst making each have its own distinct variable names. As discussed in chapter 5 that enables us to change the graph structure of generating strings so causing all derived variables to change their graph structure in an identical manner.

In string manipulation, the "value" of labels is irrelevant, the output of these functions is the arrangement of the labels, for example according to the way that points are to be joined together. Objects are formed as instantiations of prototype sets of labels, now treated as variables. This is a difficult stage as it is necessary to move from a manipulation of labels to manipulation of the *values* of variables under those labels. That is achieved by means of the Lisp function `READ` that takes

a string and returns the first word or list as a Lisp atom or list. Alternatively we can use the Lisp function QUOTE. A quoted atom or list is treated in Lisp as an object that is not evaluated. Quoted lists may be quite long and may indeed be embedded Lisp programs treated as unevaluated text. It is easy to see the power of such an ability. We exploit that property in the functions COMPLEX and OBJECT. The first converts strings into labels, the second attaches the values to the newly created variables and also creates an instantiation by creating lists of reals that can represent 2D or 3D points. This input list of reals (called a *frame*) is shifted with respect to the world origin and scaled, both according to the arguments of OBJECT. The output, another frame, is a list of reals.

EDENGEOM.LSP deals with common manipulations of geometrical entities. Geometrical operations described in chapter 5 are formed by invoking functions for vector manipulation (addition, subtraction, dot and vector products, scaling, midpoint) and matrix transformations such as translation, rotation, reflection. Again these functions are strongly typed and are checked as such by the EdenLisp interpreter. Selection of elements of lists may be made by use of PROJN, a function that accepts a list of reals, a list of real-lists, or a frame and returns the nth member. Common geometrical shapes can also be generated, *e.g.* rectangle, n-sided regular polygon.

EDENDISP.LSP contains the functions that call upon AutoLisp Commands to perform drawing operations. The main functions have been described in Chapter 5. If called from AutoLisp the AutoCAD COMMAND function must have its arguments encapsulated as a series of strings. The COMMAND function is complicated and each call must be precisely in the format that a user would enter the data at the user interface. A number of problems local to AutoCAD were encountered. For example many commands require the user to point to the entity to be manipulated. Giving the correct reference to that entity often proves tricky, particularly if a number of other operations have already been done to that entity. The most foolproof method involves identifying how AutoCAD itself references the entities. That is done by means of an entity reference called the *handle*. The handle is a unique address given by AutoCAD that accesses the details of that entity in the AutoCAD database. AutoLisp provides a number of handle manipulation commands that return the handle name, delete or edit the handle. In EdenLisp we add the handle address to the *SYM* table for each DRAWFN, *i.e.* to each definition that causes an entity to be displayed. A further advantage of the handle is the AutoLisp command ENTDEL: that totally removes the entity from both the

AutoCAD database and from the screen. ENTDEL is invoked within the implementation of EdenLisp to remove all trace of a previous definition when that is redefined. It is easy to see the benefit of that in EdenLisp: redefinition does not entail explicitly finding and deleting the previous entity from the screen. The redisplay is thus much faster than is common in traditional methods of animation sequencing for example. (Interestingly, AutoCAD "remembers" it has created those entities despite killing the handles. The command UNDO will still sequence though previous instantiations, providing a high-speed sequence through the design history. It remains to be seen how the UNDO command may be exploited for precisely that purpose!)

6.44 Actions

The Abstract Definitive Machine that is the conceptual framework of EdenLisp has the notion of *guarded action.* The implementation of the guard in EdenLisp is by a conditional definition. The definition is entered at the EdenLisp interface with a predicate in the conventional procedural form of the **if** statement,

```
Defa = if relative statement then statement else statement
```

Care is needed interpreting the meaning of this definition: the problem is the same as in the Eden interpreter. Consider the following definitions in EdenLisp

```
f = if c then x else y
g = if c then b=x else b=y
```

Here f depends on c, x and y. In the second definition, if c is true then `g` depends on x only, if c is not true `g` depends on y only. In both cases changes in x and y will affect `g` in conditions where they should not do so. If c is true then `f=x` and `g=(b=x)` in the second case.

Actions are implemented either by manipulating quoted atoms or lists and evaluating them as needed, or by using string manipulation followed by a `read`, again as a way of evaluating. Essentially an action triggers Lisp functions that generate the text of new definitions from fragments of input strings or dummy definitions. For example, a simple way to manipulate an input string is to make a function that replaces each occurrence of "`?n`" (where `n` is an integer) in the string by the element of the replacement list corresponding in position to that integer. The resulting text or dummy definition is handed to EdenLisp as an input that is effectively an alternative to the keyboard or disc file. The computer can therefore be considered to be the agent inputting the definitions. It is that idea that is developed in the multi-agent system envisaged in the ADM.

6.5 Discussion

The evaluation procedure in EdenLisp is strict forward. It traverses the tree according to the order of the dependent variables. If a variable is redefined it is important that it be added to the beginning of the 'deponits' of records of dependent variables otherwise old values could get used in evaluations prior to dealing with the new definition. The operation of adding to the front is foreign to Lisp so care is necessary in doing it and it costs some time in evaluation.

Type processing is very difficult to do if we allow the wide powers of Lisp to be used. Many Lisp operations are on lists with members that may be of any type; those operations (even the basic ones of `car` and `cdr` - returning the head and tail of a list respectively) are virtually impossible to use explicitly in EdenLisp because of the typing problem. The set of "command" operations in Lisp that deal directly with lists are therefore not implemented, although they can be accessed via embedded AutoLisp.

Type processing is computationally expensive. With the tree traversal involved with evaluation EdenLisp becomes slow in action. Care has been taken to make each process computationally as simple as possible but the sum of the processes makes updating take considerable time. Although that is acceptable for prototype activities to test the ideas of EdenLisp some work is necessary to make it useful to engineering users. Creating new functions means typing them. Some provision for that has been implemented in the form of a Lisp function `newtype(name, type, input type)` that can be invoked at the time of creating the function in AutoLisp.

The windowing environment is complicated and makes large programs potentially extremely verbose. One possibility is to consider all environments except the current one to be static or historical. That would not permit parallel environments to get changed when a common parent variable gets changed, although the task of updating will need to be redone each time a window is re-entered. The problem is to balance computational time versus space, a problem identified earlier in our discussion.

7

Experiments in EdenLisp

This chapter deals with experiments in EdenLisp that test the features of the implementation and illustrate its potential for design. A reasonable criticism of EdenLisp might be that it does not add anything to existing methods for problem-solving. The potential of EdenLisp is particularly in terms of how it uncovers relationships that are based on observation and state. However it would be a poor tool if it did not do at least what other methods can do. So we begin with some experiments with the language that illustrate straightforward design problems. We then compare approaches that use conventional programming tools and show that both economy of programming and exposure of design structures are also benefits of EdenLisp

7.1 Parametric Studies

7.11 Tumbler-Mixer Machine

Possibly the simplest way to use EdenLisp is to exploit its spreadsheet analogy to carry out design calculations. Because equations can be set out in conventional form without recourse to row-column tabular formulae peculiar to spreadsheets, an EdenLisp script reads more naturally as a design report of the design calculations. The ability to change individual definitions means that the sensitivity of particular variables may be examined interactively by a process of inspection as variables are redefined in a particular range. Such parametric studies are common in design as an interactive way of checking sensitivity, rather than using optimisation programs where the constraints have to be preconceived or where the effect of constraint relaxation is difficult to predict *a priori*.

As an example of the advantages of parametric studies of that kind, the following is one where I used EdenLisp to perform calculations for a local firm. The Company required a partial redesign of a tumbler-mixer machine that they use for mixing fine powders such as those used in the food industry.

The arrangement is shown in *fig. 7.1*. The flask contains up to 4 tonnes of powder and is inserted into a cage that is rotated about a horizontal axis. The drive shaft is attached to the cage at a position such that the cage is rotated about its vertical axis by 30°. That means that the powder is thrown from left to right as it rotates. The requirement was for the shaft to be of the correct diameter for the steady and impact loading as the powder falls about during the mixing process.

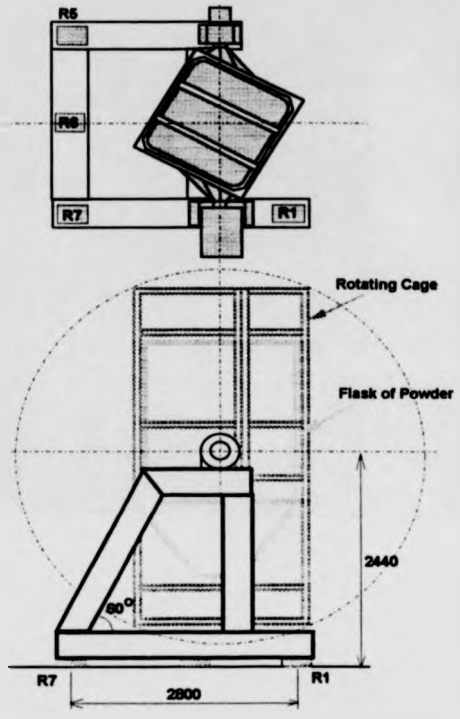


Fig. 7.1 Powder Tumbler- Mixer Machine

The EdenLisp script shown in *Appendix B* contains the formulae needed to calculate the shaft diameter. Its form is very analogous to a spreadsheet as no graphics are required for this computation. It is simply used for the parametric study. From that study it was found that, for example the impact loading was the most significant factor in the shaft design, whereas a hollow shaft could be specified with very little increase in the outer diameter. Different ways were explored to determine the machine stiffness and the effective load due to impact from falling powder.

7.12 Four-Bar Linkage

A reasonable extension of the spreadsheet analogy is to make a "graphical spreadsheet" in the form of calls to AutoCAD. We show that we can make a system that automatically up-dates an AutoCAD drawing as the user changes any of the declared parameters. The example used is the design of a 4-bar linkage. The EdenLisp script for this (*Appendix B*) is similar in structure to the tumbler design

in the previous section, but with the addition of constructions for creating geometrical objects by the method described in earlier chapters. The geometry of a 4-bar linkage appears simple in terms of the drawing but the relationships are non-trivial. For example in *Fig 7.2* given the co-ordinates for the points `J0`, `J1` and `J3`, the point `J2` is difficult to determine. The formulae for finding `J2` are as follows

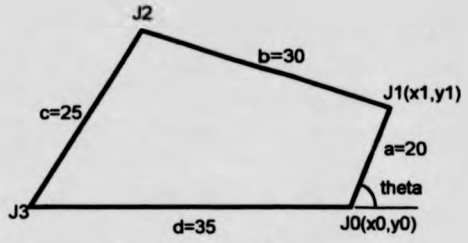


Fig 7.2 4-Bar linkage design in EdenLisp

```
L     =   - sin(theta)                    ; Computations to establish angle delta
M     =  (d/a) + cos(theta)
S     =  a*a + c*c + d*d - b*b
K     =  d*cos(theta)/c + S/(2*a*c)
sq    =  sqrt (L*L + M*M - K*K)
delta=  2*atan((L + sq)/(M - K))          ; Compute angle delta from given data

x     =  -(d + c*cos(delta2))             ; hence find co-ords of point J2
y     =  c*sin(delta2)
```

In the script nodes `J0` to `J3` are written as strings within a variable `nodes1` of type `LLstr` (list of string) as follows.

```
nodes1 = ["J0", "J1", "J2", "J3"]
nodes2 = cedge(nodes1)
```

The function `cedge` creates connections between the nodes and can set up a number of instantiations that have the same connectivities but different realisations. Function `cedge(nodes1)` on the four nodes yields the string connections:

```
(("J0" "J1") ("J1" "J2") ("J2" "J3") ("J3" "J0")).
```

Using these topological relationships the coordinates for the nodes are assigned and the strings made into variables with corresponding edge connections via the *complex* operator.

```
barb = complex (nodes2)                   ; create variables making up the 4-bar
```

The variables `J0` to `J3` now acquire their values from the definitions as follows

```
J0 = [0.0,0.0]                            ; coords of fixed pivot of driver bar a
J1 = [a*cos(theta), a*sin(theta)]         ; coords of moving end of driver bar a
J2 = [x, y]                               ; coords of driven end of bar c
```

```
J3 = [-d,0.0]                          ; coords of fixed pivot of driven bar c
```

The *object* operator computes the set of coordinates resulting from moving the nodes to be about a given origin and with given scaling factor on each dimension. The object can be realised from its skeletal structure, and displayed as single straight line segments or bars of complex geometry. The definitions are:

```
barf = object(barb,[300, 150],[5,5])   ; create 4-bar at (300,150) to scale*5
bard = Wireframe (barf, "pline")       ; draw the 4-bar as a polyline
```

Given the geometry in *fig. 7.2* we can change the definition or value of any declared variables. The corresponding coordinates are recalculated and the values of draw functions representing the lines get changed too. The effect of a draw-function change is to delete the AutoCAD handle, the address where the display information is found. The deletion actually removes the displayed entity and enables the new drawing function to display the new position without retaining the previous display. That means it is possible to have an animation by simply changing a variable sequentially. In the code that is done by having a set of values explicitly defined in turn.

An interesting feature of the display is the labelling. The EdenLisp operator `label` is designed to deal with labels so that the considerable flexibility built into AutoCAD regarding font type, shape and size can be used. At the same time the power of EdenLisp is added, in that labels can be attached to objects in such a way that their attributes may be defined in relation to that object. The code for label "BarA" is

```
Str    : labela
Lreal  : Lpta

J4     = midpt(J0,J1)
Lpta   = locate(j4,origin,size)
labela = label(Lpta, 5.0, "barA")     ; label it
```

The `label` function takes three arguments, the location of the text in object space, a list of font attributes such as height, width and justification, and the text string itself or a variable that generates a text string. With that operator, labels can be moved with the object, the text can be changed to report something about the object, the position relative to the object can be moved such that it not obscured by graphic entities. In *fig 7.3* labels `BarA`, `BarB` and so on get moved in relation to their appropriate bars.

Fig. 7.3 shows six possible arrangements of the 4-bar linkage that one might expect by varying the angle `theta`. We should be able to produce all six pictures on the display by creating different instantiations of the frame with different origins and scales. These would be distinct realisations with independent coordinate sets. However all instantiations made from the same frame will be geometrically identical since all the geometrical descriptions are based on the same definitions. Thus the picture with six objects cannot be realised unless there is an identical set of definitions for each object but on a different set of variables, *e.g.* the angle theta is a ***different variable*** on each object.

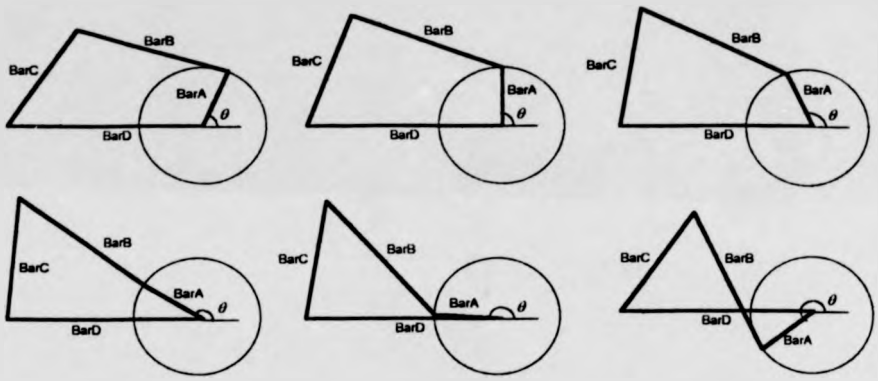


Fig. 7.3 Arrangements of a 4-bar Linkage for different theta values

The problem raised here is a fundamental one. If we wish to show simultaneous pictures then we need to have different 'windows' for holding the different values. Alternatively we can regard each Intantiation as a *suspended state*. A picture is constructed and displayed with one script of definitions and then left in that state. A separate copy of those definitions is then created as a different script and that script becomes the current state to be changed at will. This scenario is similar to that used in windowed systems such as MS Windows where a window is only active while made so explicitly.

7.13 Drawing Frame

A comparison of EdenLisp with AutoLisp is instructive. AutoLisp may be used to construct parametric drawings, an example of which is a parametric that draws a standard drawing frame and title block on a specified A-size paper. A 10 mm border is to be drawn around the paper with the title block in the bottom right-hand corner. The paper may be landscape or portrait. A difficulty is that the title block should not be directly proportional to the paper size, otherwise what is

reasonable on A0 will be far too small on A6 and what suits A6 will be too large on A0. Comparing the parametric written directly in AutoLisp with the EdenLisp program shown in Appendix B (that incidently does the task in a very different way), the AutoLisp code ran to 4 pages compared with half that for the EdenLisp. The EdenLisp script is also more flexible. For example, a single change of variable will switch landscape to portrait and the labels may be changed at will. That means that if the drawing needs to be retitled with a much longer name midway through the exercise the whole block can be rearranged to enable the longer text string to be accommodated. It may be said, quite rightly, that it is easy to include that possibility in the AutoLisp parametric. But the number of possibilities extends to all variables in EdenLisp, not just those preconceived in setting up a parametric.

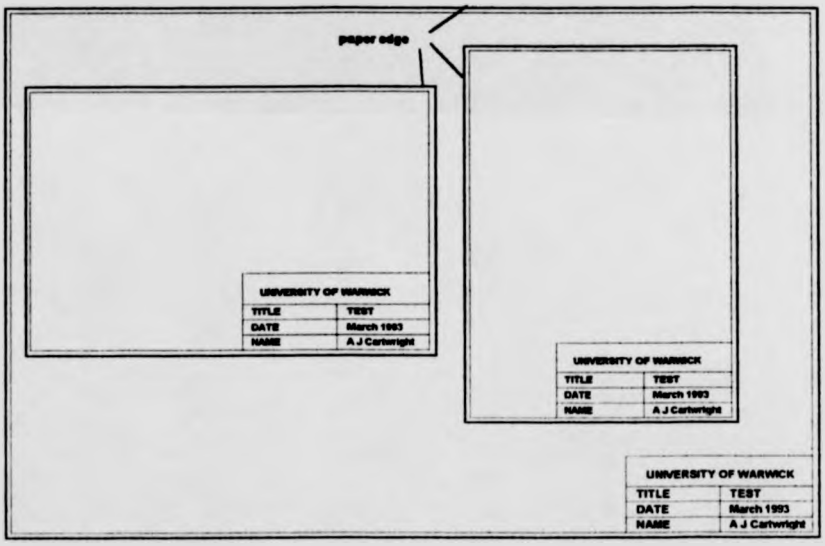


Fig 7.4 Sample outputs from Drawing Frame Program, superposed. Diagrams show A0, A6 landscape and A6 portrait (not to the same scale)

The problem of making the title block in reasonable proportion to the frame is a case in point. Realising the problem, it is possible to experiment with one or two basic definitions to arrive at a suitable formula that allows the title block to increase rather more slowly than the actual sheet size. (In fact a logarithmic relation was found to do the job!) This "sensitivity analysis" is precisely the activity that the spreadsheet is good for and so is natural for EdenLisp. *Fig. 7.4* shows outputs from the code, annotated and with paper outlines inserted for clarity.

7.14 Precision Balance

The final parametric design example explored in this section is the extension of the "wireframe" into surface modelling. Nothing new is added conceptually to the Definitive method by such extensions. Solids and surfaces are defined in terms of their generators and will be displayed by AutoCAD by simply calling the appropriate commands. Also properties of the resulting solids may be calculated by AutoCAD or by accessing information given in the definitions.

The concept used in this design is the so-called elastic hinge. [Smith ST & Chetwynd, 1992]. A sketch is shown in *fig. 7.5* of a typical notch type hinge. Because the thickness t is so small the top of the bar can be rotated forming a short-travel hinge. A monolithic construction is one in which hinges are machined from the solid by drilling holes that are 0.3 mm apart or less at their circumferences, and then joining such holes together with slots or saw cuts. An example of that construction is shown in *fig 7.6*

Calculations are based on the following notation

- t = Thinnest section thickness, (mm)
- R = Radius of curvature of the notch, (mm)
- b = Width of material at the thinnest section, (mm)
- F = Load, (N)
- λ = Stiffness of one hinge, (N/mm)
- E = Modulus of elasticity, (MPa)
- θ = Maximum angle of rotation of hinge
- L = Length of lever, (mm)
- K = Correction factor for the notch stress concentration factor
- Q_{max} = Maximum displacement of lever
- T_{max} = Shear stress on hinge, (MPa)

The following relationships obtain.

$\lambda = E\, b\, t^3 / (24\, K\, R\, L^2)$

$K = 0.565\, t/R + 0.166$

$M_{max} = b\, t^2\, \sigma_{max} / (6\, Kt)$

$Kt = 0.325 + (2.7t + 5.4R) / (t + 8R)$

$\theta = 4K\, R\, \sigma_{max} / (Kt\, E\, t^2)$

$Q_{max} = L\, \theta$

$T_{max} = \sigma_{max}\, b\, t$

$B_{max} = 6\, M_{max}\, Kt / (t\, b^2)$

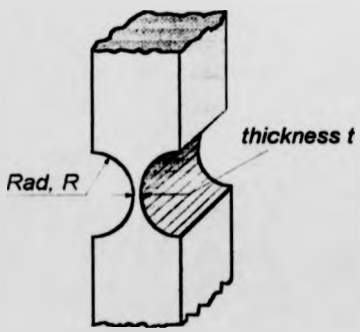


Fig. 7.5 Detail of Notch type Hinge

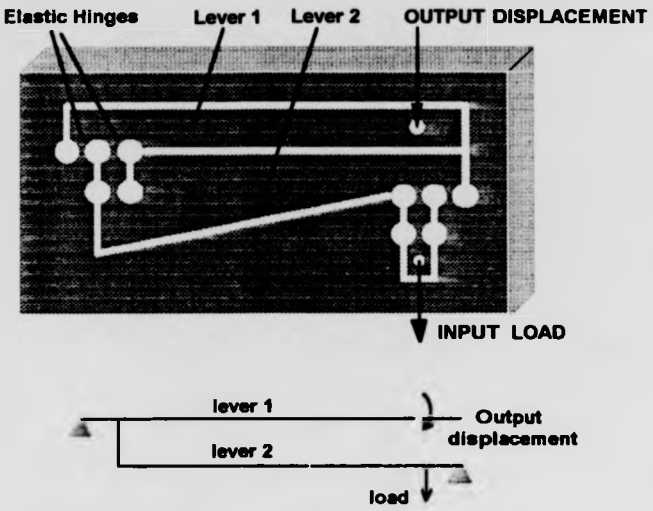


Fig.7.6. Monolith and equivalent Lever System

For aluminium we can insert some design figures as follows.

E = 72000.0
b = 5.0
t = 0.3
R = 5.0
L = 55.0
σ_{max} = 100.0
F = 400.0
K = 0.565 t / R + 0.166 = 0.565 . 0.3 / 5.0 + 0.166 = 0.2
λ = 72000 . 5 . 0.3^3 / 6 . 0.2 . 5.0 . 55^2 = 0.54 N/mm

Because of the complexity of the formulae it is difficult to decide what the important parameters are in the design of an individual notch. Numerous arrangements of the monolith construction can be made forming, complex lever systems of which one is shown in *fig.7.6.* The design of these lever systems is very much an interactive iteration, trying out many different arrangements of levers to get the most compact one that has the maximum lever ratio. The aim is to get the maximum lever ratio into the smallest space, and EdenLisp provided a neat tool to inspect different layouts. In the practical case different materials and arrangement were attempted, some of which were manufactured and tested with a view to being used in a precision weighing machine.

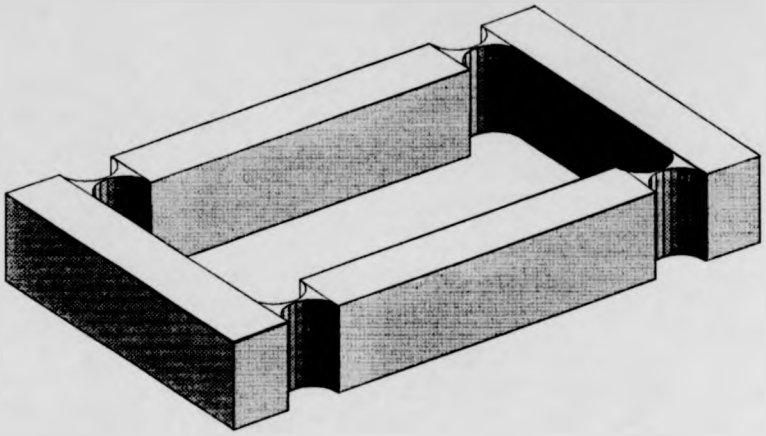


Fig. 7.7 Monolith construction in Three dimensions (with added shading)

The geometric construction of the notch was straightforward, but the display proved to be quite difficult because of rather arcane methods used by the AutoCAD system for accessing information already on the screen. The notch was constructed by using arcs and polylines, the only legal method for then "extruding" the plan shape into the third dimension. When the base shape has been constructed it is necessary to construct the extrusion vector and to refer to both the shape and the vector before the system obliges with the extrusion. Unfortunately the act of creating the extrusion vector renders the base shape inaccessible. It appears that the extrusion vector becomes attached to the base shape and so the latter cannot be addressed, even by its handle. The somewhat eccentric solution is to create the extrusion vector, point to it and then "delete" it before pointing to the base shape. The extrusion vector is then reinstated to allow the system to use it to create the extrusion itself. Because of that the shape shown in *fig. 7.7* proved very difficult to obtain, even though it contains no new EdenLisp concepts. Some artistic licence has therefore been exercised on the shading in the figure.

7.2 Patterns in Design

It frequently happens in design that patterns can be detected as underlying the main problem and these provide families of solutions to different design problems. Examples abound: parametric design yields products of a similar shape topology; designs may be built from standard elements such as gear boxes and pumps, or from geometrically similar components such as turbine blades. Often functions have to be applied repetitively to different or similar components, or different

functions affect the same set of components in different ways. The same principles may therefore be called for very different purposes. One of the benefits of the Definitive method of tackling design is that it forces one to consider those patterns, or alternatively patterns emerge from having to express the design definitively. We have already seen in the consideration of the shaft design in EDEN how the hierarchies identified as typifying design generally are exposed by the Definitive method. Now we illustrate how problems of this kind have been attempted in EdenLisp.

7.21 Analytical Graph Plotting

Arrays of entities are mentioned as pattern in design. Often the entities are identical, as in groups of standard fasteners holding the top of a machine element; sometimes there are trivial differences in adjacent entities such as in gate arrays for VLSI design. It is therefore desirable to have tools that generate dummy arrays that can be transformed into local entities as required.

The plotting of graphs provide an area for exploring arrays in EdenLisp. Arrays in this case consist of Cartesian pairs or triples, but even with such a simple array a problem of definition was identified in DoNaLD. If each point in a graph is significant then it should be defined in such a way that it does not depend upon other elements in the graph, nor should the number of graph elements need to be pre-declared, else the redefinition becomes clumsy. Giving each point a unique definition has the advantage that any point can be referenced or redefined. However the penalty is likely to be an information explosion that may not be necessary, or a tedious repetition of definitions that are virtually identical for each point. Action definitions provide a way to produce any type of definition; in particular lists can be created recursively to any size to create the required arrays to represent graphical data. As an example `Graph.Lsp` was implemented. The listing is remarkably short, given the amount of data created with unique access labels.

To make and plot an analytic function as a Cartesian graph the following sequence is adopted.

1. Create a list of length Npts
2. Write the analytic function to be plotted as a string, e.g. fn = "sin(1/x)"
3. Use mkgraf function to create the coordinates of the curve with input arguments Npts, range, varname (as string), funct (as string)

4. Draw standard axes, using range to scale the x-axis and the maximum value of the function to scale the y-axis
5. Draw the graph

EdenLisp functions and definitions that create and plot the analytic function graph are in *Appendix B*. The function at the commencement of the listing is used in the creation the array of coordinates. It illustrates the complexity of functions that underlie EdenLisp. It is frequently necessary to create such functions but the library that accompanies the EdenLisp code has been created over a period of time and covers many common applications. It is comparatively straightforward to add more functionality although it does have to be written in AutoLisp. Once the definitions are compete it is straightforward to annotate the graph with axes and labels as desired. The following figures and function were used to create the carpet graph shown in the output below. The axes are omitted for clarity.

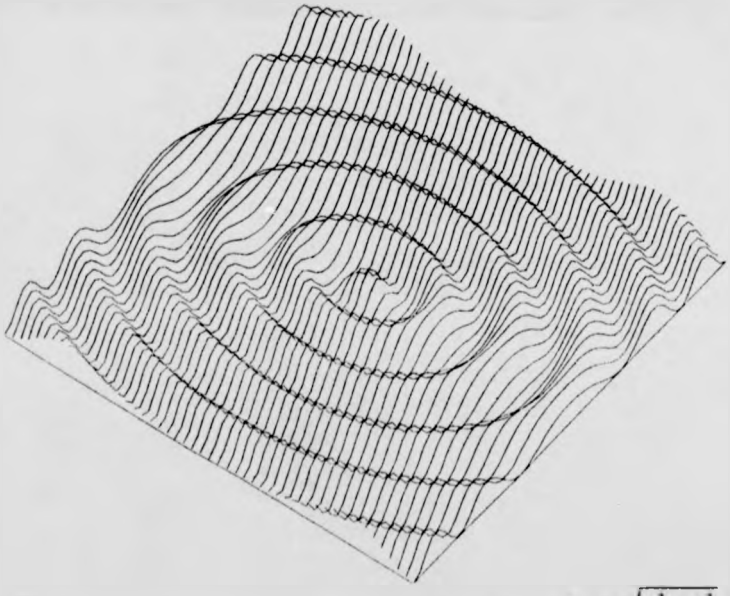


Fig 7.8 Output from Graph Lsp for the function $z = \cos\sqrt{x^2 + y^2}$

```
Npts   = 50
xrange = 40.0
yrange = 40.0
func   = "cos(sqrt(x^2 + y^2))"      ; func = graph of analytic curve

flist  = mkgraf (Npts, xrange, yrange, func)
origin = [100.0,100.0,100.0]
gpts   = object(flist, origin, [1,1,1])
gline  = wireframe (gpts, "carpet")
```

The program generates 50 points on a single value of y, varying x by the amount xrange/Npts and then repeats that for yrange/Npoints, so producing 50x50 points and 50 lines and the shape shown in *fig 7.8*, like a series of ripples on a liquid. Because the result is a single array there are no labels for each point or line. That makes the method suitable for conditions where the individual points are not of interest. If the points are interesting then we need to use actions that generate point labels. We examine methods of doing that in the next section.

7.22 A Denture Design Aid

This problem came to my notice at a seminar [Randell, 1993] where David Randell presented the software package that his team had been doing in order to help the dental profession. The software, written in Prolog, was intended to enable a dental technician to design dentures. On starting the program the user was presented with a two dimensional diagram of a standard set of teeth, such as that shown in *fig.7.9*.

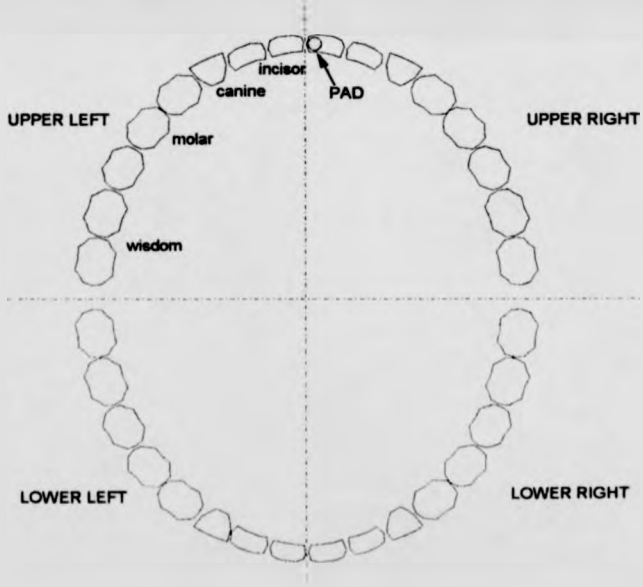


Fig 7.9 Output from the Dental Scripts

The user could then modify the diagram interactively from a menu of on-screen icons: teeth could be selected, removed, geometrically modified, moved a small distance, rotated slightly, or made artificial. The idea was that the dental technician could identify a patient's teeth profile from a pressing and then take and modify the standard diagram to enable a denture to be designed around teeth that were missing or to be extracted. Dentures are designed using standard techniques: to

hold the artificial teeth, to shape the body such that the denture is big enough not to be accidentally swallowed, to fit it comfortably in the mouth, and to do the job of biting and chewing. The rules for doing that were embodied in the Prolog program. For example, the user needs to shape the denture in such a way that forces on the artificial teeth from eating are transmitted to adjacent natural teeth, as the gums beneath the artificial teeth are unable to cope with such loads. That is achieved by bracing the denture against pads that fit into shaped orifices in the natural teeth, or by tension members hooked around the lowest point of the natural teeth next to the gums. All these members could be placed on the 2D diagram to show the design.

The problem with the software was identified during the seminar. The program was very long (described as over 20 mm thickness of A4 printout!) and although it was well structured it took considerable skill to modify the program to add new features. Constraints on the designer were built into the package to cope with various levels of user skill, so that warnings, helpful advice and prevention of impractical designs were conditional on the user's level of understanding; *i.e.* the constraints could be circumvented or removed by the expert user to reduce design time. Although the constraints were at various levels of severity they were preconceived. If new constraints were identified as desirable they could only be added by the programmer and further development was limited by the same requirement for a programmer.

The standard diagram is straightforward and can be generated quickly by any graphics system. The difficulties arise when it is desired to modify the tooth forms. The interactions to do that would be mechanical, leaving the basic reasoning and interpretation of the actions with the user. That would make the generation of constraining rules impossible. On examining the specification for constraining a design, the underlying pattern that emerged was that basic elements such as tooth, pad, hinge and plate could be represented as instantiations of particular sets of abstract definitions that give shape, position and orientation. By means of such archetypes, teeth could be placed in standard positions for quickly generating the initial set-up, but could be subsequently modified locally if necessary. Any tooth could also be formed with its own unique coordinate set but having its topology and family attributes editable in a global sense, *i.e.* wholesale changes to all the molars can be done, for example to model female, male or children's teeth with very little effort. To carry out that approach the following analysis of the abstract program was made.

Underlying pattern

level 1 Abstractions

- topology of tooth forms
- topological relationships of teeth
- definitions of components for tooth form, pad, saddle, hook, plate for denture

level 2 Archetypes

- geometry of archetype tooth forms, pad, saddle, hook, plate for denture
- positions of teeth around mouth and of pads, saddles, hooks on teeth
- number and size of teeth for different people: male, female, child

level 3 Instantiations

- instantiation of archetypes in given position and with particular rotation
- copies of instances for standard repeats such as molars
- conversion of archetype into local edition of a form

level 4 Editing

- instantiation of variant editions for odd shaped teeth, local variations, positions of caries, pads, fillings, etc.
- extracted and artificial teeth
- editing of archetypal plate for desired shape

level 5 Constraints

- Functional restrictions on editing, e.g. teeth cannot be transplanted to other quarters, hooks are tension members, pads compression members, plates must cross quarters

7.23 Using Actions

The scripts in *Appendix B* as Program 7 generate the basic tooth arrangement in the mouth and the denture plate design using both definitions and actions. The levels described above correspond to sets of scripts: each set may have actions that rewrite or create new scripts for instantiating or editing purposes. Defining forty teeth separately by their shape coordinates is tedious when an array produced from an archetype tooth form will serve as well. Editing a shape interactively is easier than inputting coordinates separately and the user should be able to do either global or local changes with equal facility. The level 1 script consists of a set of definitions that yield finite sets of labels that can be used to define any shape. The set of action definitions

```
Mkincisor = A_lst ("incisor","i","lreal",10)
Mkcanine  = A_lst ("canine", "c","lreal",10)
Mkmolar   = A_lst ("molar",  "m","lreal",16)
Mkwisdom  = A_lst ("wisdom", "w","lreal",16)
```

produces sets of definitions that are actioned as sets of labels of type `lreal` (list of real). Definition `Mkincisor` produces the following script.

```
 llreal : incisor
  lreal : i1 i2 i3 i4 i5 i6 i7 i8 i9 i10
incisor = [i1, i2, i3, i4, i5, i6, i7, i8, i9, i10]
```

The function `A_lst` attaches the string `"i"` in turn to each integer from 1 to the number given and then makes it a label, part of a list assigned to the label `incisor` derived from the first string. That label is then declared and the definition actioned. Each abstract action definition enables a new label list to be generated. Actions such as `A_Lst` are AutoLisp functions that return scripts of definitions. They are assigned type `lstr` (list of string).

Level 2 archetypes are easy to define using the pre-declared lists. All we need is to define Cartesian pairs to define a 2D representation of the shape of an arbitrary tooth, *e.g.* the following definitions are for an incisor tooth with shape indicated by the numbered diagram. Consistent numbering of the tooth form means that one can keep track of which is the outer and inner sides of the tooth form.

```
i1 = [ 17,  0]          ;
i2 = [ 15, 10]          ;
i3 = [  7, 12]          ;
i4 = [ -7, 12]          ;
i5 = [-15, 10]          ;
i6 = [-17,  0]          ;
i7 = [-15,-12]          ;
i8 = [ -7,-11]          ;
i9 = [  7,-11]          ;
i10= [ 15,-12]          ;
```

Insisor Tooth Shape

Once a form is ascribed all instantiations at level 3 can have the same form. Changes to any of the `i`-values will cause changes on all the incisors at once. If a local form is required it is simple to invoke another action definition to generate labels and use the same values of `i1, i2, ...`, etc. to initialise the local version.

The locations of teeth are defined by means of a definition of the "general mouth", an ellipse describing the shape of the mouth. The eccentricity of the ellipse is simple to redefine for different mouth shapes being done by scaling as in the following definitions for different mouth types.

```
male   = 1.0                          ; scale size of mouth
female = 0.9
child  = 0.7
mouth  = male                         ; define current mouth shape
ScM    = evalid(mouth)                ; use its real value
Rx     = 200.0*ScM                    ; elliptical shape of mouth minor axis
Ry     = 250.0*ScM                    ; elliptical shape of mouth major axis
```

The problem of large numbers of similar definition sets that was identified above applies equally when describing instantiations. One way of dealing with that is to use the technique used in the shaft analysis program implemented in DoNaLD. There new definitions were created using string substitution. For example to make a single instantiation of a tooth one needs to specify the tooth type, where it is to be located in the mouth in terms of which quarter and which position. The following script for example produces the Upper Right incisor at position `URlpos`.

```
ang    = pi/Nteeth*2.24               ; position increment round mouth
angl   = (Nteeth + 0.2)*ang           ; actual position of first tooth

llreal : URltyp                       ; declare tooth type
lreal  : URlpos                       ; declare tooth position type
frame  : URlins URldsp                ; declare object and display types
URlpos = [Rx*cos(angl), Ry*sin(angl)] ; middle of tooth profile
URltyp = rotobj("incisor",pi/2-angl,1.0) ; orientation & type of tooth
URlins = object(URltyp, URlpos, scalea) ; instance of tooth
URldsp = wireframe(URlns, tooth)      ; display the profile
```

To produce outlines of the other teeth one would need the last seven definitions to be reproduced with only slight difference in numbering or tooth type. That can be achieved by means of the Action function `A_repl` that replaces "?n" (where n is an integer) in a list of strings by the element of the replacement list corresponding in position to that integer. For example,

```
tooth_a =["llreal: ?1?2typ
          "lreal : ?1l?2pos
          "frame : ?1?2ins ?1?2dsp
          "?1?2pos = [Rx*cos(angl), Ry*sin(angl)]",
          "?1?2typ = rotobj(?3, rt-angl, 1)"'
          "?1?2ins = object(?1?2typ, ?1?2pos, scalea)",
          "?1?2dsp = wireframe(?1?2ins,?4)" ]
LL2  = A_repl(["LL", "3", "canine", "tooth"], tooth_a)
```

uses A_repl to replace each ?1 by "LL", ?2 by "3", ?3 by "canine" and ?4 by "artificial" to produce the following as a list of strings that are actioned as definitions in the usual way.

```
llreal: LL3typ
lreal : LL3pos
frame : LL3ins LR1dsp

LL3pos = [Rx*cos(angl), Ry*sin(angl)]
LL3typ = rotobj(canine, rt-angl, Z)
LL3ins = object(LL3typ, LL3pos, scalea)
LL3dsp = wireframe(LL3ins,tooth)
```

Using that technique all 40 or more teeth may be instantiated from a single definition set, of which an extract follows (for the upper right set).

```
toothp = if ScM=0.7 then "" else tooth
UR1 = A_repl(["UR","1","incisor","tooth"],tooth_a)
UR2 = A_repl(["UR","2","incisor","tooth"],tooth_a)
UR3 = A_repl(["UR","3","canine", "tooth"],tooth_a)
UR4 = A_repl(["UR","4","molar",  "tooth"],tooth_a)
UR5 = A_repl(["UR","5","molar",  "tooth"],tooth_a)
UR6 = A_repl(["UR","6","molar",  "tooth"],tooth_a)
UR7 = A_repl(["UR","7","wisdom","toothp"],tooth_a)
UR8 = A_repl(["UR","8","wisdom","toothp"],tooth_a)
```

The (cunning!) if definition for toothp enables the value of toothp to be set to nil if a child's mouth is required. In that case the wisdom teeth are missing and the mouth display is rearranged to reflect fewer teeth. The value of tooth reflects whether the tooth is artificial or natural. Similarly if mouth = female then the mouth shape is scaled by 0.9, whilst keeping the same number of teeth. The appropriate definitions are as follows.

```
ScaleA = if ScM=0.7 then [0.9,0.9]      ; scale Rxy
                    else [ScM,ScM]
Nteeth = if ScM=0.7 then 6.0 else 8.0   ; child has no wisdom teeth
natural  = "cpline"                     ; tooth is shown as a polyline
artificial = "fill"                     ; artificial tooth is shown hatched
tooth    = natural                      ; define current tooth type
```

Amendments to the position are achieved by having tooth position to be a function of Nteeth, the number of teeth per quarter. Just as the definition tooth_a is the archetypal tooth, similar definition strings may be constructed for archetypal pads, hooks and so on.

To produce the denture plate an abstract plate may be constructed around the design specification. The tasks of the denture are to replace particular teeth,

transfer load to adjacent natural teeth, cross the mouth roof or go around the lower set profile and so on. The requirement for positioning is partly defined by the position of the teeth to be replaced. The intersection of the adjacent natural teeth and the nearest artificial tooth may be determined from the centres of those same teeth on the ellipse defining the mouth shape. Thus from a natural input of the names of teeth that need replacing a general shape of the plate can be generated and a sample plate displayed.

The gap between the teeth is determined as the linearly interpolated value between the adjacent natural and the artificial teeth. This gives a positional error as the position is on an ellipse rather than a line, but that will be small. The archetype definition is `t1` as below

```
t1="?1?2?3pl = midpt([?1?2pos, ?1?3pos])"
```

For a plate covering UR3 and 4 we might have definitions that generate the four main points on the edge of the plate as follows

```
mkg12 = A_repl(["UR","2","3"], t1)
mkg34 = A_repl(["UR","4","5"], t1)
mkg23 = A_repl(["UL","2","3"], t1)
mkg67 = A_repl(["UL","4","5"], t1)
```

giving an output such as, for the first definition,

```
UR23 = midpt[UR2pos, UR3pos]
```

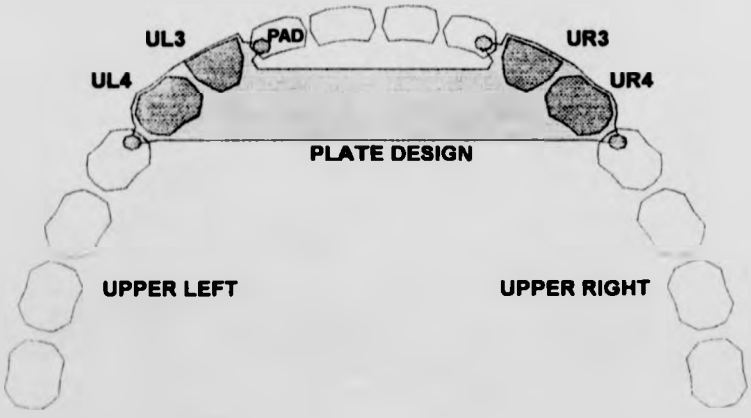


Fig. 7.10 Output of Plate Design defined on four teeth names

Other points demarcating the plate would be generated so as to follow the edges of the missing teeth, obtained from their notional coordinates from the tooth archetypes. The number of points on the plate is as many as required but experiment shows that about seven points are needed for each tooth covered by the plate. A definition to that effect will enable the right number of points to be generated. The final plate design uses the AutoCAD command "`Offset`" that takes a polyline as input and produces another polyline offset to one side or other as requested. A sample plate design is shown in ***fig 7.10***. It may be modified to produce the desired shape by appropriate re-definition of the points that make up the definition of `plate`. The pads that enable the plate to be attached easily in the mouth and transfer the load are positioned in standard positions on adjacent natural teeth and assume that the natural teeth can take the extra load.

To make the identification of the tooth positions easier it is best to use labels in the definitions that correspond to the tooth positions, and also to number the label points making up the tooth form in a logical manner to enable the designer to pick off the right points. That process is aided by the fact that dental practice has names for each side and feature of the teeth.

The addition of help and constraint guided functions is an exercise in educational software explored in a later chapter.

Action functions lead not only to an economy of programming but also to the wholesale changing of definitions by a single definition. It is these functions that enable the implementation of agent-oriented programs with the power to make changes in scripts.

8

Design Management

Understanding the roles and interactions of different agents in design should be of interest to those who manage design in industry. In this chapter we return to the problems of integration introduced in section 5.1. We consider the role of the design team in design management before discussing how definitive script manipulation can be used to develop an approach suited to a multi-agent scenario.

8.1 The Design Team

8.11 Development of Design Management

It is known that the design phase influences some 80% of the product price whilst the design activity itself is a small cost to that product (Helldén, 1987). It is therefore not surprising that there has been markedly more management interest in the design process in recent times. While that might be thought to be a good thing, the initial results have not been encouraging. Managers have tended to make management decisions about design without a proper understanding of the process. Particularly in the traditional industrialised countries such as UK, US and Germany, companies have not regarded designers as being in line management and so there has been a poor history of design management. One symptom has been that the introduction of computer based design processes has been hampered because managers are somewhat chary about allowing access to Company financial information. That has meant that designers could commit firms to inappropriate expenditure or not take advantage of favourable trading conditions with friendly firms. Conversely, management policy might get formulated without reference to the traditions built up by designers with customers and contractors. For example, "bread and butter" jobs that keep firms going in lean times are chopped by management as giving a poor contribution to profits. As management has moved from keeping designers at "arm's length" to forming design teams, it has proved to

have been a rather rough journey. The tendency has been to try to keep the compartmentalised approach but to reorganise to permit greater use of technology such as CAD.

Evidence of that management development is contained in a survey reported by Wilfred Miller [Miller, 1989] and concerns the implementation strategies for CAD in West Germany over the 5 years 1984-89. He reports that 40% of medium sized companies surveyed who used CAD had no clearly defined management of CAD implementation. The consequences in those cases were that CAD managers were not suited to their tasks, CAD was not being integrated with manufacture, there were sharp divisions between CAD and manual methods, and there was very little training. In the remaining 60% of the survey there were interdepartmental groups set up by senior managers to implement CAD. Significantly however the managerial emphasis was computer orientated rather than process or design orientated. As a result there was no proper authority structure to carry out a co-ordinated policy on CAD. Although CAD was successfully introduced, with better drawings and reduced iteration between design and manufacture, there remained conflict, particularly at the interface between design and manufacture. In only a relatively few cases were companies appointing managers with engineering design qualifications who had high responsibility for integration, particularly at that interface between CAD and CAM.

The problems of implementation of CAD highlight the fact that industry still suffers from Islands of Automation. Many activities in the product generation process are still having to be done by hand or by different groups of independent agents. The design-manufacture interface is a particularly acute one, as has been identified technically by many working in CADCAM but is latterly being realised by managers too.

The management difficulties identified here arise from an historical practice of specialisation, where product generation tasks are grouped such that the responsibility of one function is with one phase of many products. Each function was dealt with independently, with little interaction with other functions; each produced documents that provided the input to the next phase as a kind of "message passing". Dealing with feedback of negative comment from other functions led to a long time-span from conception to production. That time lag becomes apparent if one compares specialisation with an alternative approach: to integrate task responsibilities from concept to production along a single product.

In the latter case the time lag is much shorter as noted by Swedish investigators [Wærn, 1986] where such "vertical" integration is the norm.

It was the attempt to get the best of both specialisation and integration that Forward or Concurrent Engineering has been developed. Design teams accept joint responsibility for a single product and different products have different teams, albeit with considerable overlap. That makes use of specialist skills whilst limiting feedback to iteration rather than criticism. "Message passing" still exists, but tends to be more constructive. For concurrency to operate each member of the team has to have up-to-date information of the state of the design. That makes for many meetings and multiple copies of documents appertaining to the design. Design management in such circumstances becomes a significant problem because of the potentially damaging prospect of copies not being in step with one another. Chris Voss and Graham Winch identified these problems with their observations on organisational links for CADCAM implementation. [Voss, Winch, 1989]. Use of a common data base, where data is accessible across functions is essential. Putting different groups of people in physical proximity also helps. However the significant feature identified by Voss in his empirical study of a motor vehicle company was the need for a co-ordinating person with an overall knowledge of the CAD system with multiple functional and integrating skills.

These observations concerning integrative methods are highly significant in connection with the ideas described in this thesis. If we can identify and co-ordinate all the agents in the design process and have them working on a single document, analogous to simultaneous working on a physical prototype, then we have the potential for true concurrent engineering. We can then address the real difficulties of concurrency, namely the identification of agents and how they interrelate. Those problems are extremely difficult and are the subject of current research programs. [Beynon, Cartwright, Joy, & Godfrey, 1993]. The following section describes the background and current progress in identifying who the agents are in a prototype design and how a multi-agent orientated definitive system might be implemented.

8.12 Agents in the Design Process

The term *agent* is used in Definitive methods in the manner described in chapter 4. However it is useful first to have an intuitive understanding of agents in the design process. In management terms an agent may be thought of as being in charge of a task or tasks that can be carried out largely independently of other tasks in the

overall process, as for example in a particular line sequence of a PERT or critical path analysis chart. At the top level an agent is a person with functions normally found in industry. At that level the members of a design team are the agents. For example, we can group the functions in the design of a mechanical engineering product into agent areas as follows.

Market research
Ascertaining the market demand
Forming the preliminary product or systems specification:
Defining essential and desirable functions,
performance, target price, environment, appearance, service and maintenance, product functional life, product run life.

Research and development
Examining the physical principles to be exploited and coming up with novel processes and product possibilities, or refining ideas generated in response to user enquires

Mechanical design
Developed from the specification in consultation with R&D. Functions include
Preliminary design - establishing the "design space" relations
Feasibility studies - using tools such as a graphical sketchpad to layout ideas
General Arrangement - arranging the design layout with reference to mechanical analysis.
Materials selection with respect to parametric property requirements
Detailing - using databases with catalogue and other data of standard features and components

Mechanical analysis
Compares product requirements (the demand) with the performance of real materials and systems (the supply). Analytical tools might be:
Power analysis: power flows in the system, interactions of effort and flow systems
Geometrical modelling and properties
Finite element analysis: stress, strain, thermal, static & dynamic stiffness, vibration

Electrical and Electronic Engineering
E.g. design of drive and control systems and devices.
Links with mechanical and other hardware. Heat transfer analysis.

Systems:
Computer system hardware: input and output devices;
Software for low level support and high level control.

Industrial Engineering
Aesthetics, environmental issues, anthropometrics

Manufacturing:
Process planning: group technology families. process features, tolerances.
Manufacturing technology: process: machine tools, tooling, jig and fixture design, materials handling, scheduling
Assembly requirements - assembly sequence, robotics programming

With that diversity of independent interests, interactions can occur in uncoordinated ways, resulting in designs with particular weaknesses and strengths. The co-ordination identified by Voss needs to be done by a design manager with an understanding of the current state of interactions. The design manager would perhaps coordinate the design team interaction in the context of regular meetings to give mile-stones to the design process, to agree strategic decisions that constrain future developments in areas in the concurrent system such as

- the specification of the product,
- common requirements of team members,
- subsequent requests to modify the prototype.

The support that the computer can give the design manager in may be correlated with the extent to which representations in the design process may be successfully integrated. We have already pointed out in earlier chapters how difficult that process is with current tools. The most obvious difficulty is that of data representation. When processes are specialised we can, and frequently do have a situation where data internal to the specialisation has a representation that is peculiar to that discipline. Only in the message passing is data presented in a common format; although even then the format may be governed by the specialisation that other functions simply have to learn, rather like some English assuming that "the French can jolly well learn English if they want to communicate with us". Thus CAD has its own internal data representation (in fact numerous different ones!). Standards such as IGES help to provide a common output but that output is not necessarily helpful to the other stages in product generation. CAD to CAM has already been identified.

The way pointed up by IGES is that it may be possible to translate ideas or data into useful formats, albeit with some loss of definition in many cases (*e.g.* translating 3D to 2D). An integrated model can operate using existing systems provided such translation takes place. For example some standard CAPP (Computer aided Process Planning) techniques use expert systems and an extensive database of proven process plans to generate suggested manufacturing plans for a given drawing using tolerances to imply particular processes and functional properties to imply materials and heat treatment. The input to such systems still needs to be in machine readable form such as IGES code. If the information is less precise, *e.g.* from a scanned machine drawing then the intervention of human skills may then be necessary.

A considerable difficulty with translation is to have two-way message passing. Translation is normally one way: the drawing imported into the Word-processor from a CAD system cannot be passed back for amendment in the light of examination in the later process. Ideally we would want to have a common data representation for an integrated system. But even then we can get into difficulties. If we seek an integrated model that should support processes that are normally peculiar to a specialist function, the representations have to serve a multiplicity of functions. It may then be that there will be problems in separating the roles of different agents.

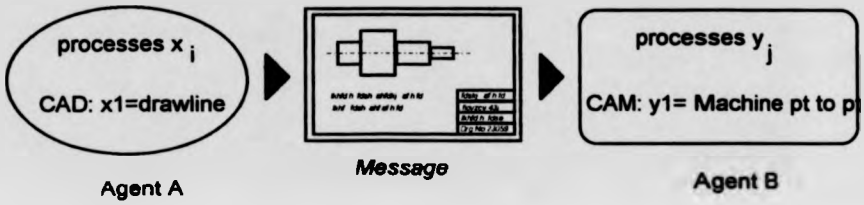


Fig. 8.1 Data Representation and message passing

In *figure 8.1* we consider data representation between two respective functions of Agent A, a designer, and Agent B, a production planner. Information peculiar to agent A (for example, process x_i where x_i might be to specify a line) would apparently not be of interest to a reader of the drawing produced as the output message since it relates to the method of producing the line on the drawing, not what that line represents. Similarly agent B, would have "private" ways of dealing with machine tool operations, part of process set y_j. In a fully integrated model we would wish support

- an intelligible interface for agent A to influence process y_j and for agent B to influence x_i
- the protocol for A to influence processes y_j and vice versa.
- For some information to be not being fully specified *e.g.* for prototyping

In the "private" method the computational approach may be regarded as largely born out of batch processing. In parallel processing there needs to be a more fundamental approach to the computational modelling; one that allows for communication, interaction and concurrent action. We need to have a computational model of the product that permits the kind of private activity that characterises much of an agent's normal work whilst also allowing access to other

agents, albeit within the constraints of the specialist's discipline. That model might allow different representations of the data such as graphical image, functional model, physical or catalogue data and properties, manufacturing process planning schedule, simulation of behaviour during manufacture or in ordinary operation.

This is the style of programming that is espoused in this thesis. Agent oriented definitive script manipulation allows the agent approach to be developed meaningfully in management terms. The definitive manager might then correspond to Voss's suggested "Design Manager". The difference would be that the function would have to include fluency in the definitive method as well as competence in the management of design decisions. That is clearly a major hurdle whilst operating at code level but in the development of the idea the system will eventually be simplified by suitable user interfaces.

8.2 An Agent Oriented Approach to Design Management

Within a single management function we can identify a multiplicity of tasks with the same pattern of specialisation and integration scenarios described for the whole process. So we reduce the scope of the problem in order to highlight how progress can be made in terms of formulating agents for sub-tasks and their interfaces with other agencies.

If we take a single function such as the mechanical design of a component, [Wærn, 1986] differentiates the tasks that have that degree of independence. He quotes the following breakdown of activities involved in design, from a study of three companies in Scandinavia.

	% of Total Time
Administration and planning	10
Retrieval of information	11
Problem solving	18
Computing	6
Drawing and changes	32
Assembling information	8
Checking	6
Other	9

Ave. time for different activities in design work, from [Wærn, 1986]

That analysis provides the basis for breaking up the design process into domains under the control of independent agents. Suppose for example we take the biggest

task, "drawing and changes". Here we may have a number of different people occupied with different aspects of the design, each with particular areas that are totally independent, but all with some commonality at their interfaces. If we return for our example to the shaft design problem promulgated in the previous chapter we can identify Agent A, responsible for analysis of the shaft design, Agent B a detailer and Agent C the manufacturer. In overall charge we have the design co-ordinator.

Now the analyst and the detailer may wish to experiment with different materials for their own reasons. The first wants a particular strength and stiffness pattern for the ideal material; the latter wants the cheapest and most readily available standard material within or close to the property range specified. All have excellent reasons for choosing their particular material and each choice has different effects on the design. Thus each needs to have the liberty to explore on the current prototype the consequences of different choices. Clearly at some points there will exist numerous scenarios, exactly as in normal design developments. The task of the design co-ordinator is now to adjudicate at those places where a decision has to be made on what constitutes the "current prototype" and what are local variants. Those choice points are the milestones of the design identified above. So we can write families of definitions that represent the different patterns of work that are currently allowed. Those constitute the constraints on the design. The constraints would not necessarily address which agent is allowed to choose the material but maybe who is responsible for a particular issue from that choice. The issue might be who determines the maximum deflection of the shaft. That makes the decision scenario demand driven rather than supply driven. Clearly the choice is ultimately one of policy, directed by the specification of the initial design.

The computational model of the current prototype can be thought of as a "virtual prototype" a term that has echoes of virtual reality and so seems apt for this purpose. The virtual prototype, or VP, begins its "life" as the script of definitions that picks up the initial specification of the product deduced from the decomposition stage outlined above. As it gets thought about in the creative ideas stage, the VP may run off in a number of directions as each agent gets to work. If ideas flow a number of possibly independent "solutions" may be suggested. In *fig 8.2* those are shown by the different agents branching off the main VP. In the real world these ideas would be recorded (or at least the less outrageous ones!) in a design folio. In Definitive modes these ideas would be represented as separate scripts of definitions, each describing a solution in terms of the initial specification.

To make sense of these scripts, each agent keeps its set separate from the principal VP until an idea becomes accepted at the milestone points. If a set is accepted then that becomes part of the Virtual Prototype to be worked on from that point on. If the design is concurrent then all agents agree to use that common VP. The variants are not discarded but remain in the design folio as alternative routes in case the current VP proves infeasible.

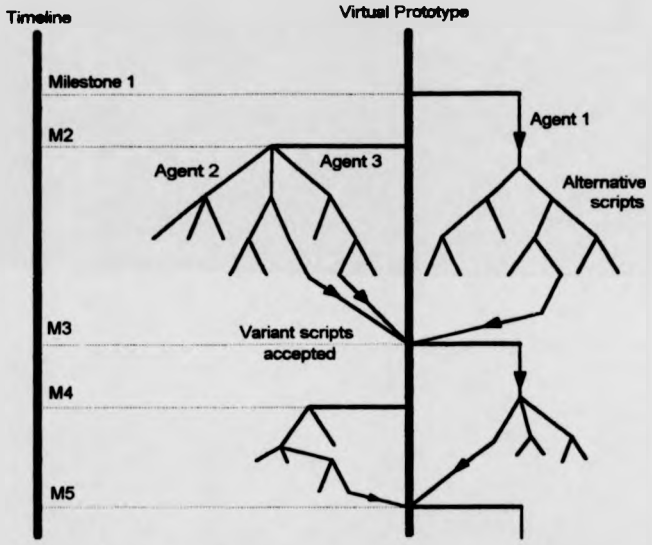


Fig. 8.2 Design of Virtual prototype by concurrent scripts

Different designs can be developed independently until sufficiently defined to make a selection. Thus each agent can have a number of design histories in a folio and argue for the one that is to become the VP for development at the milestone point. Further, the interactions of various parameters and groups of parameters become apparent as the design proceeds and it becomes necessary to group such interactions into separate scripts. Since the parameters will in general be interdependent it is necessary to identify within a script those variables that can be changed without affecting other scripts, those that can be changed only within constraints imposed by another script and finally those that would impose constraints upon variables or groups of variables in other scripts. It is here that the notion of agents needs to be developed.

With different anticipated decision patterns, families of definitions can be included or excluded from the current prototypes by simply having functions that do that by

means of setting their guards on. If guard A is set then Agent Detailer would have the responsibility of providing the definition script for that area and also putting up the constraint pattern that permitted limited access for further experimentation. In that way the analyst would not suddenly discover that the material had become cream cheese! Some of these issues are discussed in [Beynon, Cartwright, Yung and Adzhiev, 1994] where the idea of virtual prototype is explored further.

The snag with recognising patterns of decisions is the risk of an explosion of alternatives. It needs careful organisation so that decision patterns are defined strictly on the basis of particular product specification. In our example if three agents have decision making access for say the material choice then, with both supply and demand scenarios we could end up with 6! = 24 choices merely to make one script part of the prototype. Unless automatically generated it becomes a serious piece of programming to anticipate all these scenarios.

The approach using Actions described in the last chapter could provide a way to structure such choices to prevent the combinatorial explosion. Decisions could be delayed or made more abstract by the generation of definitions from text. This is the subject of further work.

9

Design Education

The demands upon students of engineering design are intense. They need to understand analysis of engineering systems, synthesis of functional solutions, and to be familiar with many computational tools. They are challenged now to put product generation and systems design into the contexts of international, environmental, economic and political spheres wherein the design has to operate. All that has to be carried out in a climate of increasing competition and reduced life cycle time. The design process is rapidly becoming computer-based rather than computer-aided, by which is meant design must be carried out at a level that cannot be done other than with computer modelling. The growing complexity of products and the trend towards concurrent engineering in design all reinforce that trend. What would be of great help in an increasingly computer literate student body is for the very process of using computer systems to encourage a proper understanding of the design process.

In this chapter we explore ways that definitive methods may be used to help students and others towards computer-based modelling and towards self-teaching by interaction, animation and prototyping.

9.1 The Educational Context of Design

9.11 Historical Background

Historically, innovation has been thought of as a kind of amateur game. In both Britain and America, myths about inventiveness such as "necessity is the mother of invention" and "Watt and his steaming kettle" tended to reinforce that idea, despite the fact that innovation was fed by solid scientific discovery and technological change. The separation of innovation from science led inevitably to the separation of industry from academia. As science grew so it became the province of specialists

separating into departments of physics, chemistry, geology and so on. The result was that the brilliant achievements of gifted mechanics and engineers such as Maudsley, Nasmyth and Whitworth were based upon basic training and apprenticeships.

When by the middle of the nineteenth century Britain noticed that other countries were more successful technically, the answer was sought by introducing Engineering into the Universities. The effect was not as great as expected! After the 1st World War the government took over much of the industrial research because of industry's reluctance to innovate and the growing gap between science and industry. The pattern of modern education was set. The authors in [Burns and Stalker, 1986] put it like this:

> "Two major changes have occurred in the social circumstances affecting the production of innovations. First, industrial concerns have increased in size: greater administrative complexity has brought in a wide range of bureaucratic positions and careers. Their positions make it imperative that innovation was seen to come from within not by newcomers.
>
> The other change has occurred in the form of institutional relationships within which innovation had been possible. The familiar and social circumstances typical of the eighteenth century provided the ease of communication necessary for the major synthesis of ideas and requirements that introduced the early revolutionary inventions. In the nineteenth century new institutional forms introduced barriers between science and industry. By the twentieth century the new and elaborate organizations of professional scientists has been matched by one of technical innovators into groups overlapping teaching and research institutions, Government departments and industry"

The institutionalisation of design has not proved to be helpful because of that separation of product and process. It led directly to the idea of Engineering Science, the ultimate separation! Design was interpreted as a branch of analysis. Indeed the author's own experience is of a "Design" course in which a clutch is "designed" from its description. What was asked for was the calculation of the clutch plate size, an analytical problem with the design taken out.

9.12 Learning the Design Process

In the last twenty years Design teaching has undergone a renaissance in both government and academia, following various reports such as those by Bullock on Academic Enterprise and by Feilden on Engineering Education. Enormous effort has gone into trying to understand the design process and to find better ways of

teaching it to students. However "standard" approaches to teaching design still appear to collude with the idea that there is a sequence of activities that will lead inevitably toward a "correct" solution. A typical student text has the suggestion that the first three stages of the design process are called "Problem Finding"

> "1. **IDENTIFY** the fundamental need to be satisfied
> 2. **DEFINE** the precise problem arising from that need
> 3. **PARAMETERS**: state the constraints within which any solution must fit" [Starkey, 1992]

The desire to structure design has carried over into computer aided design. Students need to know that many computer tools over-constrain the designer. Those kinds of tools are designed to solve particular problems and the inputs must be precisely defined. Indeed there are those who see design as needing such constraints. Some of those making computer aids for manufacturing would like to constrain the designer by limiting the features that can be part of a design, for example to those that can be made with current technology. The danger in limiting innovation is clear. That kind of bottom-up approach may be helpful in many detail design problems but one should recognise that one of the reasons for a design aid being made at all may have been that it could be made with the computational tools available, not that it was necessarily the most important. (As often happens in life, we try to solve the problems that look tractable and ignore the intractable ones and hope they go away!)

The design process is much more elusive than implied by these "steps to a solution". It is interesting that practising designers do not identify with any of the so-called design process descriptions beloved of academics. A recent (mischievous!) comment by Allan Gardam, Chief Mechanical Engineer at Pilkington Optronics, was that the best description of the design process is represented by a single block diagram.

That comment is supported by work done by [Kelly, *et al.*, 1986] who comment

> "As we reviewed the various theories and models, we began to realise that in almost every major innovation of recent times each functional phase is linked in some way to the others: every phase in our block diagram has lines connecting it to and from every other block in the diagram. Instead of a linear-sequential picture .. we had a plate of spaghetti and meatballs!"

All is not lost however. We can identify some important ingredients of design, particularly at the conceptual stage. The most important of these has already been discussed at length: namely observation and experiment, for which the EdenLisp is designed. Trial and error are the very stuff of design, and of science itself. The act of finding out has still that charm, often indeed thought of as mere playing. Sir Hermon Bondi makes this observation in the Foreword to [Michie, 1986].

> " I myself was involved in space affairs when in April 1970, a serious malfunction in the Apollo 13 mission to the Moon led to great anxiety for the safe return of the crew. By a rapidly devised brilliant strategy, the crew returned to earth safe and sound, albeit without landing on the Moon. When I expressed my astonishment at the speed the solution had been found, I was told that the staff at Mission Control had been spending their time playing games with the equipment and that rescue from disaster was one of the games! Our play instinct is always something to be fostered."

Play is of course not totally unstructured. As one finds something that amuses or interests it is investigated more thoroughly, an approach that has its counterpart in design. It is that which researchers into learning have found to be most significant in gaining and retaining knowledge. Taking one extreme, the effort required to retain small amounts of "nonsense syllables" was found to be excessive because there was no relation to prior knowledge. In real-life, as we have found in the discussion on Minsky, knowledge is "chunked" into percepts that relate common observations. The question is then how new knowledge gets chunked. [Wærn, 1989], in discussing general learning principles, shows that the most successful learning situations are top-down; they arise from linking new knowledge with prior knowledge and then being able by reflection to discriminate and then to generalise. Discrimination consists for example of a child seeing cows and horses and not calling them both "bears", the only prior concept she had for large animals. It is the process of seeing what is different and what is similar in the new situation. Generalisation is the chunking stage, associating similarities.

Both discrimination and generalisation refer to *declarative* material. The results of learning declarative material are always expressed declaratively. We have to fetch the material directly from memory and reproduce them. *Procedural* knowledge on the other hand has to do with associating knowledge. It is a much more difficult learning operation with three stages. First there is the cognitive stage: the declarative knowledge, then an associative stage, putting knowledge together in sequence, and finally the autonomous stage where the knowledge becomes chunked. The first stage is easiest and learning is fastest. Learning then drops off rapidly.

The lesson from this discussion is clear. A top-down, declarative approach is most fruitful both for learning *about* design and for learning *to* design. We can therefore benefit from some of the research into the design process namely the notion of hierarchical decomposition.

9.2 Learning to Design

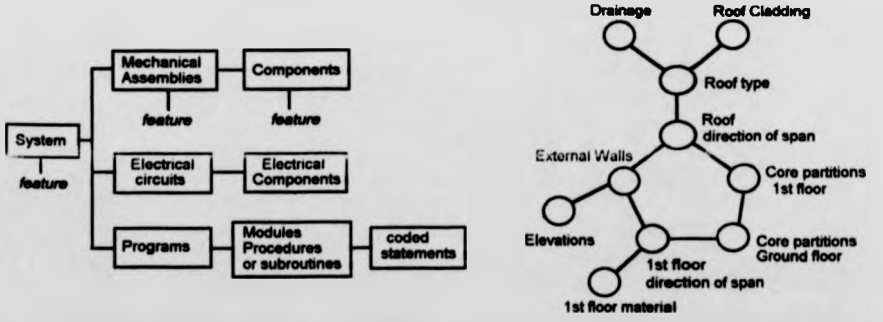
9.21 Hierarchical Decomposition

Decomposition is the first stage in the top-down approach, as has already been discussed in chapter 2. Ullman suggests how it helps to structure one's thinking in arriving at the requirement.

> "In general, during the design process, the function of the system and its decomposition is considered first. After the function has been decomposed into the finest subsystems possible, assemblies and components are developed to provide these functions. Thus a hierarchy of mechanical is shown in the top row of *[the figure reproduced as fig 9.1a]*. Also shown in this figure is one further decomposition of mechanical objects" [Ullman, 1992, *chapter 2*]

Sometimes the problem is not that easy to structure hierarchically in Ullman's way. For example, compare Ullman's nice hierarchy *fig 9.1a* with the cyclic problem in *fig 9.1b* that [Cross, 1989] raises, where the problem is of a particular house design detail identified by [Luckman, 1984].

> "Architects identified five decision areas concerned with the directions of span of the roof and first floor joists, and the provision of load bearing or non-load-bearing walls and partitions. Making a decision in one area had implications in other areas that had implications in further areas, in one case coming full circle."



a) Decomposition of Design Disciplines from [Ullman, 1992]

b) Decomposition Structure of House Design, from [Cross, 1989]

Figure 9.1 Decomposition Problems

However even here, Cross argues, cycles can be avoided by strategic choices and despite such difficulties most design problems can usefully be decomposed in a hierarchical manner.

We have shown in the examples in chapter 7 that the definitive method is both declarative in form and admirably suited to hierarchical decomposition. Initial statements of a design problem can be written in EdenLisp as definitions in quite vague terms such as the following.

```
CarEngine = f(EngineType, TransmissionType, maxPower, maxWeight)
EngineType = choice_of(diesel, petrol_injection, petrol_carb)
MaxPower = powerRange(max, min)
```

Each of the definitions becomes the starting points for the specification, initially without any defined variables. Functions would need to be defined but may be quite simple selection functions such as `choice_of`. (In EdenLisp there would have to be type declarations, but that too becomes a useful conceptual exercise, thinking about what the parameters would be in a design and sorting out the important from the less important or downstream variables.) As the specification gains detail certain parameters will acquire values or a range of values that define the "design space" delineating possible designs.

What is instructive is that the specification of the design in definitive terms requires the designer to decompose the tasks and suggest related tasks and possible solution spaces. As those definitions take the form of lists, they are open ended and invite addition and redefinition - an interaction that is vital at the initial formulation of the design problem. Second, that decomposition helps to identify the possible agents interacting in the design process. Third, the designer also begins to identify abstractions in the design. In order to arrive at sufficiently general definitions of particular relationships it is necessary to think quite hard about the patterns that underlie the design spaces. That is apparent in the dental plate design described in some detail in chapter 7. Patterns are identified that make possible alternative designs easy to generate, sets of relations become separate scripts quite naturally and so enable agents to be teased out. We therefore conclude that EdenLisp provides a structuring method for design that is natural, interactive and generic.

9.22 Design Folio

Having ascertained the specification and the main hierarchies of sub-problems the designer proceeds, according to [Starkey, 1992], to the next phase called "Problem solving".

" 4. Create ideas for alternative solutions
5. Evaluate each of the created ideas
6. Isolate the preferred solution
7. Implement that solution."

Again the phraseology seems deceptive. The casual reader might think that by putting heading 4 under Problem Solving the author is intending to show that given sufficient preparation in steps 1 to 3 the designer can converge on a "solution". It sometimes happens that no solution is possible to the problem as put. It would not be a useful exercise, for example, to design a lathe that can machine a high precision spindle of 0.3 mm diameter to 0.001 mm, whilst on the same machine be capable of machining a shaft of 600 mm diameter. In such cases a revision of the basic requirement is called for. It is necessary for the student to regard a cyclic or iterative approach to be the norm at any of the stages, rather than design being seen as a sequential stream of processes.

A further ambiguity in the idea of design being problem solving is that it may be perceived in terms of analytical tractability. In all but the most trivial of designs, because of the infinite variety of choices, the known information is small compared with what is unknown: rather like having 100 simultaneous equations and values for only 20 of the 100 variables. Any analytical model of the design is therefore going to be limited. We explored this point in conceptual terms in chapter 2. It is essential that the student cultivates an approach that bears these difficulties in mind. It is in that frame that we suggest that the Definitive method, perhaps in the form of EdenLisp can help.

We showed in chapter 8 that we can create with Definitive methods a computational object as a Virtual Prototype. Using the decomposition model described above we can develop the design by analogy with the non-computational approach. The designer tries out a number of different ideas, developing some of them to a degree that shows their feasibility. Those ideas will go into the design folio. In a similar way, the student could develop a number of definitive scripts as candidates for the Virtual Prototype, these being stored in an equivalent "design folio", probably in the form of a library of files in a directory. That computational design folio is not simply a set of library files on the same topic, as for example the collection of generics in PADL-2 or the storage of partial solutions in the SDRC I-DEAS CADCAM software. The key difference is that in those systems the composition of partial solutions has to be done by the user providing the connections interactively. In EdenLisp the connections can be made by means of

guarded actions. Although the user's reasoning process may be the same in each case, it is made explicit on the Virtual Prototype. And because relationships are based upon real-world observations, the reactions of the VP to any further alterations are more related to the way a physical object would behave.

From the educational point of view the discipline of needing to tease out the inter-relationships between the different ideas is exactly what one would like to see made explicit. The student has to decide upon the relationships to be made and in order to do that is forced to "think aloud" about the design problems. Furthermore the decisions needed at the milestones are those the designer needs to make in progressing the design in non-computational methods. The application example of the dental plate design shows how that pattern-making process operates in practice. The use of EdenLisp in education would therefore encourage the student to adopt a realistic approach to decision making and encourage greater insight into the design possibilities

9.3 Parametric Studies

9.31 Sensitivity Study

Analysis of designs has long been the domain of the computational engineer, that phase being the easiest to automate. The value of EdenLisp is that it enables experiments to be carried out that may or may not be analytically tractable. Designers are always having to deal with partial solutions where it is necessary to "suck it and see". That process is easy with EdenLisp as it will ignore partial solutions that cannot be solved and present to the user those relationships that do have values. Because of that users can play around with values of parameters in design relationships in order to get a "feel" for the way that those parameters behave. Different values of variables in the domain of the designer are quickly entered and the effect observed. Manipulating EdenLisp scripts can enable sensitivity analysis to be more flexible than conventional optimisation techniques: observations may be made on the effect of incremental changes in parameters and also the effect on the constraints imposed on the optimisation.

By way of example we investigate the optimisation procedures for the selection of sizes for a compression spring suggested by [Siddall, 1982]. Loading is static compression and the spring ends are assumed to be closed and ground flat.

Notation

N = Number of active coils (usually end coils are inactive, so N = no. coils - 2)
D = Mean diameter of the coil (mm)
d = Diameter of the wire used for the spring (mm); usually a preferred standard size.
G = Bulk Modulus (MPa)
S = Maximum shear stress of spring material (MPa)
F_{max} = Maximum working Load (N)
l_{max} = Maximum free length (mm)
D_{max} = Maximum coil diameter (mm)
d_{min} = Minimum wire diameter (mm)
F_p = Preload compression force (N)
C_f = End Coefficient of the spring (= 1 for parallel ends with one fixed, one free)
δ_{pm} = Maximum allowable deflection under preload (mm)
δ_w = Deflection from preload position to maximum load position (mm)

1. Criterion Function

The optimisation criterion is that of minimising the volume of the spring wire used.

$$U = \frac{\pi^2}{4} D d^2 (N+2)$$

This criterion is subject to a series of 8 constraints Φ_i as follows.

2. Constraints

1. Strength. The shear stress in the spring must be less than the yield shear strength of the spring material, S_{shear}. The stress has two components: a shear force on the cross-section of the wire, and torsion of the wire. The total shear stress in the wire is expressed in terms of the loading and a spring index C_f, a function of D/d. The stress constraint Φ_1 is

$$\Phi_1 = S_{shear} - 8 C_f F_{max} \frac{D}{\pi d^3} \geq 0$$

2. Deflection constraint

The stiffness of a coil spring K (N/mm) is

$$K = \frac{G d^4}{8 N D^3}$$

The deflection (mm) of the spring under maximum static load is $\delta = F_{max} K$. The spring length under load F_{max} is 105% of the solid length the spring. The free length is given by

$$l_f = \delta + 1.05(N+2)d$$

The deflection constraint is

$$\Phi_2 = l_{max} - l_f \geq 0$$

3. Wire Diameter constraint

The wire diameter must not be less than the minimum specified.

$$\Phi_3 = d - d_{min} \geq 0$$

4. Coil Diameter

The outside diameter of the coil must not exceed the maximum

$$\Phi_4 = D_{max} - D - d \geq 0$$

5. Coil Winding restriction

The mean coil diameter must be at least three times the wire diameter to prevent it being too tightly wound.

$$\Phi_5 = C - 3 \geq 0$$

6. Preload deflection

The preload deflection must be less than F_p/K and so the constraint is

$$\Phi_6 = \delta_{pm} - \delta_p \geq 0$$

7. Total deflection constraint

The combined deflection must be consistent with the free length of the unloaded spring

$$\Phi_7 = l_{max} - \delta_p - \frac{(F_{max} - F_p)}{K} - 1.05(N+2)d \geq 0$$

8. Specified deflection

The deflection from the preload position to the maximum load position must be

$$\Phi_8 = \frac{(F_{max} - F_p)}{K} - \delta_w \geq 0$$

We do a parameter study by entering the equations as EdenLisp definitions, and the constraints as `Phi1`, `Phi2`, *etc.* as definitions with violation messages. Drawings of the spring under the conditions applied are easy to do by associating the variables with appropriate geometrical models. Possible EdenLisp script and some results follow.

```
;;; EDENLISP Exercise in Optimising compression spring dimensions
;;;
;  Definitions of relationships
Vol   = pi^2*D*d^2*(N+2)/4
K     = G*d^4/(8*N*D^3)
Lf    = dw + 1.05*(N+2)
delpm = Fp/K
```

```
; definitions of constraints
Phi1  = S - 8*Cf*Fmax*D/(pi*l^3)
Phi2  = Lmax - Lf
Phi3  = d - dmin
Phi4  = Dmax - D - d
Phi5  = C - 3
Phi6  = delpm - delp
Phi7  = lmax - delp - (Fmax-Fp)/K - 1.05*d*(N+2)
Phi8  = (Fmax-Fp)/K  - dpm

constraint1=if Phi1>=0then print"Phi1-ve, Stress too high"
constraint2=if Phi2>=0then print"Phi2-ve, Spring solid with no load"
constraint3=if Phi3>=0then print"Phi3-ve, spring wire dia too small"
constraint4=if Phi4>=0then print"Phi4-ve, Coil dia exceeds design max"
constraint5=if Phi5>=0then print"Phi5-ve, Coil dia too small"
constraint6=if Phi6>=0then print"Phi6-ve, Preload too great: spring
solid"
constraint7=if Phi7>=0then print"Phi7-ve, Combined deflection too great"
constraint8=if Phi8>=0then print"Phi8-ve, Load too great: spring solid"


;;; Results of two sets of values of variables d, D, N and material

Strength: S                 1100      Stiffness: K              96.02
Elastic Modulus: E        205000      Deflection: δ             46.24
Bulk Modulus: G            80000      Free Length: Lf          259.39
Force: Fmax                 4440      Delp                      13.87
Length:Lmax                  355      CF                         1.58
Wire diameter: Dmin            5
Outer diameter: Dmax          75      Optimisation: Volume   73629.59
Preload: Fp                 1332      Constraints to be >0
Preload def: dpm             150      Phi1                       6.28
Deflection: dw                32      Phi2                      95.61
End Coefficient: CE            1      Phi3                          2
var: d                         7      Phi4                         47
var: D                        21      Phi5                          0
var: N                        27      Phi6                     136.13
constant: pi              3.1416      Phi7                          0
Ratio  D/d: C                  3      Phi8                       0.37


Strength: S                676.69      Stiffness: K              97.35
Elastic Modulus: E        207000      Deflection: δ             45.61
Bulk Modulus: G            80000      Free Length: Lf          354.62
Force: Fmax                 4440      Del: p                    13.66
Length:Lmax                  355      CF                         1.55
Wire diameter: Dmin            5
Outer diameter: Dmax          75      Optimisation: Volume   182991.0
Preload: Fp                 1330      Constraints to be >0
Preload def: dpm             150      Phi1                       2.31
Deflection: dw                35      Phi2                       0.38
End Coefficient: CE            1      Phi3                          4
var: d                         9      Phi4                         38
var: D                        28      Phi5                       0.11
var: N                      30.7      Phi6                     136.34
constant: pi              3.1416      Phi7                          0
Ratio  D/d: C                3.11      Phi8                      -3.06

"Phi8-ve, Load too great: spring solid"
```

Optimisation may be carried out as a batch process using numerical methods such as Siddall suggests in his text (*op cit.*). The advantage of interactive study of sensitivity is that the student can observe what happens to the optimisation function as different parameters are manually varied. Changing the values of variables in EdenLisp just involves redefinition, so the designer can quickly study the behaviour of important parameters and can just as easily change constraints. The tables show the results of varying the material strength, wire diameter and coil outer diameter. By checking what is happening to the constraints at the same time it is possible to see whether the constraints themselves are reasonable. When the value `Phi8 = - 3.06` is obtained in the second table, indicating a constraint violation, we can not only to vary the main parameter to get us out of violation but also see whether the specified deflection from pre-load position to maximum load position δ_w needs to be changed. It is difficult for a preconceived optimisation analysis to anticipate those kinds of adjustments to constraints; and for the student to know what to do with a result of an optimisation analysis when constraints are not examined in that way.

9.32 Parametric design

A second way that EdenLisp helps in parametric studies is in discovering relationships that represent combinations of parameters in a design. In our search for methods of analysing the shaft described in chapter 4 we showed that the relationships between segments of the shaft could be expressed in a hierarchical way by series of matrices that were peculiar to the geometry and material properties of each segment. Whilst identifying the components of the hierarchy, it became clear that certain combinations of parameters were important, as for example L/EI (symbols are respectively segment length, Young's Modulus and Second moment of area of section). What we can then do is analyse those combinations across different designs and different materials, maximising or minimising the parameter combination according to the design criteria. These parameter combinations are probably best expressed dimensionlessly, and many studies have been undertaken to help students understand and use for example particular material property combinations. Sensitivity studies of these parametric combinations is equally easy to do with EdenLisp

9.4 Self-Teaching

9.41 Animation

One method of animation in CAD is to take snapshots of the screen as slides in sequence and then show them quickly to create the illusion of animation. Alternatively a simple shape may be displayed on the fly, deleted and re-displayed in the new position. That second method is intrinsic to EdenLisp inasmuch as the automatic recomputation that takes place on redefinition not only overwrites previous parameter values with the new values, it also deletes any graphic entities that are changed as a result, and reconstructs the object in its new geometry.

An example of a self-learning script (written in DoNaLD) was discussed in principle in chapter 4, namely Bow's Notation for bending of beams. In the diagram of *fig 4.7* reproduced and modified below as *fig 9.2* the three components of the notation are shown: beam loading, the vector diagram, and the polar diagram. These are connected by a graphical construction as follows.

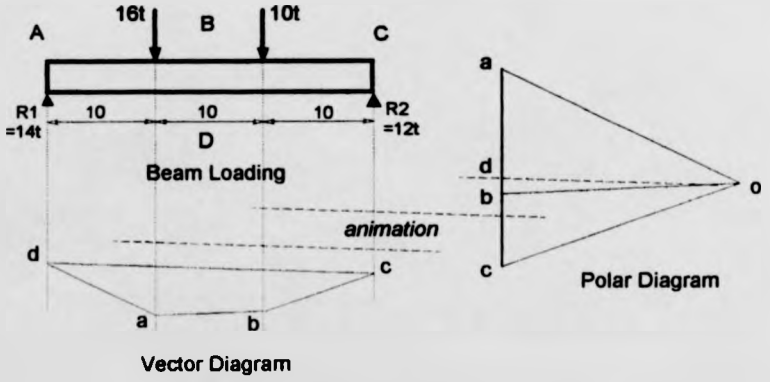


Fig. 9.2 Bow's Notation Animation

Bow's notation:

On the Beam Loading diagram, label spaces between the forces. So, A is in the space between the 16t load and the reaction R1; D is between the two reactions and so on.

Construct the polar diagram from vectors that sum the known loads. So, vector ab represents the 16t load between A and B, vector bc the 10t load between B and C. In the diagram the loads are vertical but they need not be. Select any point not on the line as the pole o and join oa, ob and oc.

Translate vectors oa, ob and oc to fill the spaces labelled A, B and C projected down from the beam loading diagram in the area designated as vector diagram. Join c to d representing the resultant of the vectors. Translate vector

cd back to the polar diagram to pass though pole o. Line od cuts the vector abc at point d. ad and dc represent reaction vectors R1 and R2, measured from the diagram as 14t and 12t.

In seeking to teach Bow's Notation it is useful to animate the construction of each stage of the diagram. In the DoNaLD code for example, once the line dc is constructed in the vector diagram in the manner described, the code causes an animation of the translation of dc to the polar diagram. Displaying intermediate positions of the translation is done by redefining the vector dc for selected points along its path. The system re-evaluates the vector each time, drawing it in the new position after deleting the previous one. Unlike the picture in *fig 9.2* the vector is only seen instantly in one position but it appears to "move" across from the vector diagram to the polar diagram. Similarly the vector ad and dc detach from the polar diagram and translate to the reactions R1 and R2 and their magnitudes get written as labels.

As explained in chapter 4 DoNaLD is rather a clumsy tool for that animation and EdenLisp is more versatile because of the way it uses AutoLisp functions. To show that difference, a construction was made of a small bench vice in three dimensions. Animation of the movement of the handle may be achieved by a series of redefintions of its angular position, causing the vice grips to appear to move. The application to mechanisms and loci are obvious, and as observed in chapter 4 the student has full control over all the variables, including the presentation of the display itself.

9.42 Authoring

EdenLisp can make use of AutoCAD's own system to help the student learn to use AutoCAD itself (a rather nice case of self reference!). The method is extensible to any situation where annotated graphics and text are required. We thus have a possible authoring system.

The method is to use AutoLisp functions accessed by EdenLisp that use the AutoCAD system of Programmable Dialogue Boxes, described in their Customisation Manual [AutoDesk, 1992]. The Dialogue Control Language (DCL) provided by AutoDesk allows one to program pop-up boxes, typically to display text or graphics and to provide buttons, sliders, highlighters, lists, toggles. Whatever is designed to go into pop-up boxes may be under the control of EdenLisp. Text that helps the user to see what is going on can be popped onto the

graphics screen at the position where it does most good, for example to point to some aspect of the display that needs explanation.

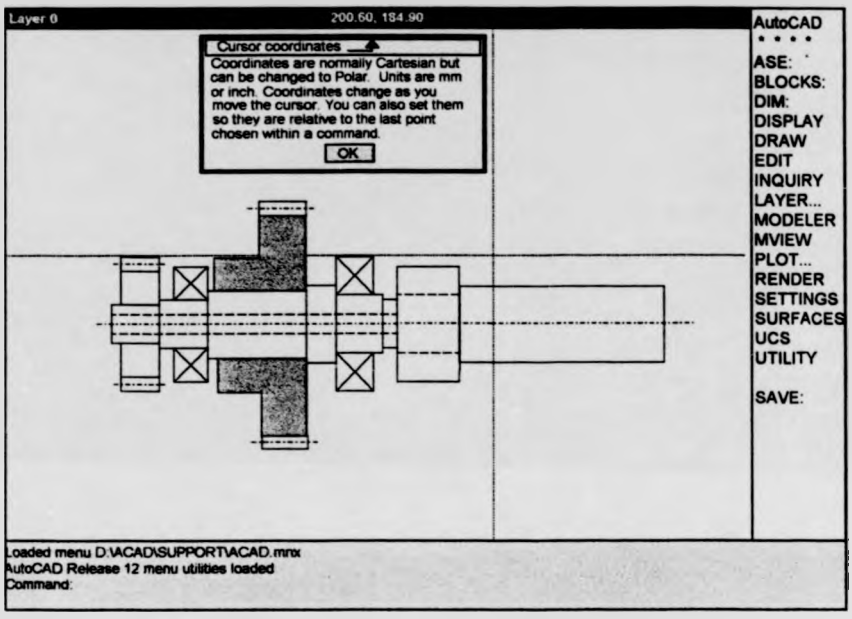


Fig 9.3 Example of using Dialogue Boxes from EdenLisp

Fig 9.3 illustrates the point. The dialogue box is created by putting the text of the box in one definition and then calling the DCL function to define the position and shape of the dialogue box. In the diagram the box has a single OK button that has to be clicked with the mouse to proceed. Other buttons, sliders, *etc.* can be similarly programmed and the results of touching those buttons can be monitored resulting in changes in the current state of the scripts. As the user changes the screen by selecting different commands it is possible that new help files can be popped up. Currently the method is driven as an embryo teaching aid, showing that it can be done in principle. Further development it necessary to make it a usable package

10

Discussion

This thesis is primarily concerned with a novel computer tool to aid the conceptual stage of engineering design. During the course of the work new insights were gained into the design process, particularly in appreciating the roles of observation and experiment, and the parts that agents play in design. Definitive methods have the property of latent state that allows them to faithfully model experiment. The implications of having a state based system to aid design have been explored during the development of EdenLisp, the Definitive programming system linked with CAD. In this chapter the benefits that EdenLisp brings to the conceptual design process are discussed. Also the stage that notation has reached is reviewed and a prognosis is presented on its further development.

10.1 The Thesis

10.11 Understanding Conceptual Design

The concept of "computation as experiment" has evolved during the period of the work carried out to support this thesis. Indeed, the investigation into applying definitive methods to the design process via EdenLisp has become a major vehicle for insight into the nature of the conceptual design process.

The link between the design and the designer is very close at the conceptual stage. It is difficult to separate "the design" from the mass of thoughts, ideas, trials, experiments and hunches. It may ultimately be a hopeless task: the design may be inchoate at the time one is trying to disentangle it. The idea of having computation reflect that stage of "chaos into order" is very attractive. The many design and program examples scattered through this dissertation are proof of the way that "real" design can be handled by definitive methods with profit. It is not claimed that any of these examples could *only* be handled by definitive methods. Indeed it would be easy to assert the contrary view. All of the examples can be written in

other programming languages. The uniqueness of the method lies in its ability to help formulate, as well as solve problems: to "think aloud" with them, to test out ideas, to frame relationships and try them out, to search for pattern and abstraction and experiment with generalisations that seem to point to a richer class of solutions than solving *this* design problem.

In chapter 3 we noted that the physical world is essentially understood by people in terms of observations. The precise mechanism for combining and experimenting with observations is still under contention. Minsky's theory, discussed there, supports a kind of grouping of observations into "frames". A similar notion of "chunking" discussed in chapter 9 is indicated by psychologists as typical of the declarative approach to learning, where people link new experiences with old and make discriminations and generalisations. The abilities to record observations and discriminate between them seems to correspond to important activities going on during conceptual design. A Definitive script supports these activities in two ways: it faithfully records observations by means of definitions and it allows the incremental examination of the interaction of those observations by redefinitions.

A single definitive statement such as `Car_colour = blue` is a metaphor of an observation or characteristic property that should find a counter-part in the physical world. A selection or grouping of such statements constitutes a kind of abstract object, perhaps corresponding to Minsky's frame. It is that selection process that matters when forming concepts, rather than what the actual constituents of the object are. The latter seems to have only a passing relationship with the *process* of conceptual design. If one concentrates only upon what that object might look like, or worse, tries to encapsulate that object, then one misses the point. The object at that stage is only a provisional set of observations to be tested against the real world and constantly to be open to detailed alteration or radical reorganisation.

The testing process is the second feature mentioned for Definitive scripts. When a definitive statement is redefined (as `Car_colour = silver` for example), the effect of that change is experienced (computed automatically) in all other definitions that depend upon the changed statement. The important feature of that change is that it is not predetermined by the existing scripts of definitions. Any change in a script automatically creates a new state that mirrors the change made. If the script faithfully records observations of the real world, by means of relationships found by experiment, and the change also is consistent with those

observations then the new state will be consistent with physical reality. It is the ability to perform incremental changes that gives rise to the notion that each definition carries a set of *latent states* - states that can be obtained by redefinition in a consistent way. The way that one checks on the new state after a single change, rather than accumulating a series of "logical" changes, appears to accord with our human perception. We prefer to be constantly checking out our perceptions against reality, a procedure endowed with the name "non-monotonic reasoning" by some logicians.

In earlier chapters we explored the contrast between *form* and *content* and showed that content cannot be captured by any set of symbols. With EdenLisp one can explore the "content" of a design by recording as many observations as seem to be consistent with its specification, including perhaps observations that do not appear to be significant. Not all of the observations that are recorded may be needed in the design. Indeed some that are needed may turn out to be missing. At the conceptual stage of the design it is never really clear how the observations will interact in the product of the synthesis. What the designer is seeking is a novel combination of observations, a kind of pattern seeking or imposition of pattern that effectively groups observations. It is only in interaction that one can begin to explore the implications of linking particular patterns of observations. Inspection of the redefinition of the car colour mentioned above yields an unexpected content: "silver" can be interpreted as a colour or as a material, with very different perceptions of its meaning. That kind of perception would be impossible to automate and emphasises the importance of interaction: any computational tool that provides for that will certainly support conceptual design. Interaction becomes progressively more significant as the design task gets bigger. As the number and complexity of the observations grow so the "experiments" one can do by redefinitions within the scripts also grow.

10.12 Computational Experiment: The place of EdenLisp

The Definitive method was not originally developed with engineering design in mind. It is really a way of computer programming that differs significantly from existing forms in that it hides those aspects of computation that do not belong to the process being modelled. Computational operations such as iteration, recursion and evaluation are not important to *what* is being modelled, they belong more to *how* it is being modelled. An analogy might be seen by watching someone crossing the road, glancing to each side to check it is safe to cross. She seems to know exactly how fast to cross and not appear to be dodging cars. If asked she would

not be able to describe *how*, but would certainly say *what* she was doing. The process going on in the brain is hidden at a lower level. In a similar way the definitive script is evaluated transparently to the user, and re-evaluated automatically each time a definition is added or changed. A further property of the notation is that it does not need to sit on top of a conventional programming language. Unlike higher level languages such as so-called 5th-generation languages, there is no intrinsic need for that. It is feasible to have it definitive all the way down to processor level - indeed it is conceivable that a definitive processor could be designed.

The crucial development in definitive methods was the definitive evaluator (Eden) by [Yung, 1987], implemented to enable algebraic relationships to be processed and hence permit experimentation that was algebraic in nature. The desire to have a similar tool to experiment with graphics relationships led to the line-drawing definitive notation, DoNaLD. That work demonstrated the possibility of applying the paradigm to computer graphics and thence into design. Work began on a 3D version of DoNaLD called CADNO [Stidwell, 1989]. It was then realised that to develop the graphics further would entail entering computer graphics research, and that that would severely constrain the development of definitive notations. The decision to consider the design process, without further development in graphics, was made possible because of a significant property of definitive notations: they are "impure" in the sense that both functional and procedural forms are permitted in definitions. That makes a notation able to call other programming forms that exist as procedures or functions. Thus DoNaLD makes use of existing graphics libraries; interfaces to the screen make use of graphics and screen routines under X-Windows.

Given that graphics handling systems can be linked to definitive notations, CAD systems are better candidates than simple graphics libraries, especially as many engineering design aids began life with CAD. We have seen that the choice of AutoCAD with AutoLisp has proved to be beneficial because of the declarative form of Lisp. EdenLisp simply calls appropriate AutoCAD functions, both for drawing and for revising drawings.

Setting up EdenLisp was a major task and might have got in the way of the primary aim of aiding the conceptual design process in engineering, except that the implementation issues actually helped to illuminate aspects of that process. For example input definitions must be stored in a symbol table for evaluation purposes.

That task could have been done, by analogy of "definition as observation", simply by means of a linear list. It is more useful, however, to record them hierarchically in terms of connections with such abstractions as lead to their instantiation as objects. The user should be allowed to make the connections and the symbol list should keep those connections in a way that allows for extending and editing the observations in the appropriate context. The flexibility of the Unix directory structure makes it a useful model and that has been implemented in EdenLisp, even though it has raised all kinds of questions about the links that now must be made between the "windows" or "environments" in the structure.

The ways used by the mind to select and structure observations are also likely to be used to synthesise new ideas. Groups of observations become "chunked" and then can themselves have higher level structures with other chunks. Those higher level structures will be complex, involving interactions with other, autonomous, objects. It is that thinking that led to the idea of autonomous objects being like agents and thence to agent-oriented programming. Programming of that nature is still in process of development and the Abstract Definitive Machine (ADM) described in chapter 3 (§3.42) is being elaborated to cope, aided by the ideas decried in this work [*c.f.* Adzhiev, Beynon, Cartwright, Yung, 1994*a*]. The ADM was originally devised as a methodology to structure definitive scripts [Beynon, 1990]. It is not itself a programming language; it is more like a pseudo-code. As the ADM sorts out the interactions of agents, it helped to structure the symbol table in analogy with Minsky's frames. Abstract "objects" within scripts written in EdenLisp can be developed interactively as the design evolves. Scripts can even be generated by autonomous agents provided the interactions with other agents are covered by appropriate constraining definitions.

The requirements for constraints to be clear and for progress in the design to be monitored inspired the notion of the "virtual prototype", a computational model that structures definitive scripts in a way that gives behavioural characteristics similar to an engineering. That is the state of the set of scripts is the state that is currently reached in the design. That idea still needs some working out, but it has the potential to support some very important features of the design process, including interaction of different agents exploring separate substates for the prototype simultaneously. The most significant issues needing to be addressed with concurrency are associated with decision-making and constraint management, but while these remain subjects of further work there is little doubt that the principle is a sound one.

10.2 Achievements

10.21 Interaction

It became clear during the course of this work that the inability of computational tools to model the "content" of symbols can be offset by a more explicit partnership with the user. For that to happen interaction has to be a fundamental part of the tool being created. Definitive methods differ in that important respect from other programming paradigms. Many standard programming methods have by their nature a static or predetermined set of states which the user activates by appropriate inputs. With a definitive notation the program plus data has current state, but with an infinite number of other states. Proceeding to another state is by incremental action by redefinition; it is not pre-programmed. The current state of the script of definitions *is* the design rather than being a *description* of the design.

The way that EdenLisp has been constructed constrains the user to modes of interaction that actually support the engineering design process. The need to declare the type of variables encourages the user to think carefully about what is being attempted. For example, it may cause one to consider topology before geometry, so stimulating the formation of more abstract representations of the design problem or sub-problem. That, in turn, may suggest more useful ways of tackling the basic design, or it may cause a whole family of possible design scenarios to be considered. EdenLisp's use of strings to generate abstract labels is increasingly recognised as a useful device for creating new or generic variables. (It is noteworthy that MatLab uses just such a method for creating its object environment.) EdenLisp is able to deal very effectively with the creation of character based lists. Some of the string handling ideas derive from trying to translate DoNaLD statements into Eden in such a way that Eden can be employed both to create and to interpret the statements being created. Lisp is better structured to deal with that kind of translation.

Interaction via EdenLisp may be at different levels. At the top level it is with the set of definitions: at the second level it is by writing or generating functions and procedures to be used in those definitions. Writing new functions involves some knowledge of AutoLisp, which is a disadvantage since most engineers are unfamiliar with Lisp. To counter that the functions can be constructed to be as generic as possible and so interaction can be by means of libraries of functions that can be built up to the level where few new ones are required in a particular case. That would mean that an engineering user would need to create few if any special functions to carry out a particular design task.

10.22 EdenLisp CAD Environment

The most significant achievement in this work is the implementation of a definitive notation within a CAD system. The CAD and EdenLisp environments together provided a platform for the development of the definitive method itself. The link between design and geometrical modelling has been complemented by linking the specification of the design with the initial thinking about that design. The formation of the design specification can be done using the EdenLisp notation. That means that as flesh is put onto the design geometrical models can be formed and checked at every stage. The idea of "what if?" that is inherent in the method is very apparent by the way the display gets updated at each redefinition. It is most illuminating to see the display of the dental arrangement (described in chapter 7) showing the tooth forms as they get modified by a single change of variable. The animation enhances the effect and stimulates the designer to try other possibilities, so increasing the likelihood that the design is the result of a decent search though the feasible solution space.

The exercise of creating library functions for topology and geometry showed that it is possible to use AutoCAD commands embedded in the function. The user can call precisely the geometry required but then play with the definition whilst retaining the design constraints. If a polyline is to be modified into a spline then the symbol table retains the connectivities that the node data of the spline has with other entities. The same data can in principle be used to convert the display from wireframe to a solid model - a step that in conventional CAD is quite difficult. A bonus that comes with the use of a CAD system is that the geometrical data is stored in two ways, one in the symbol table and another in the CAD database. Although that creates some redundancy, unnecessary duplication is avoided by connecting the two forms. EdenLisp takes advantage of the references (handles) AutoCAD has between its database and its display. Display information is processed in AutoCAD whereas structural or design information is in EdenLisp - a useful division of labour that also emphasises the role of EdenLisp as a design tool.

10.23 EdenLisp Implementation

A number of issues were recognised in the implementation of EdenLisp. Apart from the symbol table discussed above, the main issues were speed, size of code, the need for an intermediate code (as between DoNaLD and Eden), the typing of variables and the links with CAD.

Speed and Memory

Problems of speed and memory size were addressed by trying to write all Lisp functions as simply as possible. In designing a function, a recursive form was used to start with as that usually keeps the length of program text to a minimum. Where recursion caused problems by too deep nesting then a tail-recursive form was tried. Tail-recursion keeps the stack depth to one and is also the form that is most easily re-implemented in iterative form if speed tests showed that to be desirable. The code is therefore as compact and as fast as the AutoLisp allows. Tests also showed that a significant improvement was always gained by compiling the AutoLisp code.

Typing

Typing creates a discipline in programming that forces the user to think in a manner that is tied to an EdenLisp way of thinking. The types constructed in EdenLisp are 'character', 'integer' and 'real 'together with list of each and list-of-list of each. The latter allows for the development of topological and geometrical operators. More generic types such as 'symbol 'and 'list 'may be required at times to deal with odd functions and conversions but these are not usual.

The implementation of typing of variables in an untyped language like Lisp proved exceptionally difficult. Several particular problems can be highlighted. The first concerns functions that are created by the user when constructing definitions in EdenLisp. There is no separate EdenLisp function generator in the way that Eden has Eden functions without recourse to the underlying C language. Functions have to be written in AutoLisp and pre-declared and typed, to establish them as legal EdenLisp functions, before they can be called. A library of many useful functions has been built up and pre-declared in EdenLisp, but a new user function has to be entered into the EdenLisp structure so that the parser recognises it. Initially, the user had to actually enter the type data into the EdenLisp program code. That unsatisfactory solution that has now been rectified by a function that does the task in a more transparent way.

A second problem arose in dealing with what is called "quoted" atoms and statements. Those have a peculiar but extremely useful status in Lisp. A quoted atom or statement is not evaluated by the Lisp interpreter, so a quoted atom returns its name, not its value. Lisp can store any list, including what amounts to program text, as an unchangeable object until such time as it is wished to use that content that is quoted. When EdenLisp program text input by the user contains quoted statements as part of a definition it is desirable that they are dealt with in

the same way as Lisp does. In these cases parsing is difficult to do as the lexical analyser wants to identify all that is in the input text, but that device is implemented as part of EdenLisp.

Notational Consistency

It is perhaps inevitable that the work on constructing EdenLisp commenced before the conceptual ideas had been fully sorted out. Ideas were implemented and then changed. It often proved difficult to reconcile the old and the new. Where possible the samples of code that have been discussed in the text have been cast into the most recent notation. Earlier versions of EdenLisp contain significant differences, some of which remain in the program code as possible re-implementation areas should there be a need. Some of the changes may seem rather cosmetic. For example it may be better to have a different symbol for "=" in the definition assignment. That is because the assignment of a variable to a definition is rather stronger than the "=" sign implies. Eden uses the symbol "is" to indicate that the definition is one that gets updates automatically if any dependent variable gets amended. The Eden notation allows the use of "=" when assigning a constant (as an indication that it never needs to be re-calculated). In EdenLisp it was decided that the use of "=" for all types of definition was justified since all definitions are scanned during evaluation, whatever form they have.

Some other ideas tried in the early stages have left their mark. The methods used to structure the symbol table for the windowing system deal with the programming problems of global and local variables rather than properly addressing the relationships an object might have with its constraints. That problem remains as a subject for further work.

Most recent developments in EdenLisp are attempts to address the problems of agent interactions. Those are what has led to the changes in the ADM in order to express how the different agents in a system relate to one another. Proper programming of these ideas in EdenLisp still lies in the future

10.3 Comparisons

10.31 Conventional Approaches

An important issue with any new method or tool is its potential. Is the method actually going to do more than existing methods in terms of engineering design? Is it worth the bother of learning arcane methods of construction? It does seem from these investigations that the Definitive concept differs significantly from existing

methods, particularly with respect to the way it expresses state. It is difficult to define what is meant by "state" in computational terms, but there is no doubt that "the current state of the interaction" is a notion that expresses a design idea in the construction of a prototype.

At its heart conceptual design appears to be concerned with making new connections within the set of mental conceptions that the designer has obtained by experience. That experience is fed by other people's experiences but ultimately all experience is based upon experimental observations of the real world. "Conceptual design is a kind of negotiation between *what we believe to be true* and *what we observe to be true"* [Adzhiev, Beynon, Cartwright, Yung, 1994*b*]. In generalising from experience one tries to create *integrity* in what is expected. Objects will have patterns of behaviour and that behaviour will be constrained in ways that are believed to be typical or expected. The characteristic of experiment is *immediacy:* the experimenter is concerned with what is observable now, and what incremental changes might be possible. Changes are not preconceived: one can perform whatever experiments one likes on real objects and observe what happens.

In computational terms, integrity is well covered. Objects are represented in Oriented Programming and patterns of behaviour by formal specifications. State transitions are pre-arranged by providing the ability to change values of variables. Traditional Computer Science gives well specified, precisely circumscribed behaviours of reliable computing devices. Definitive methods give a direct correspondence between values in the computer model and observables in the external world.

One can go further. Computer Science tends to work by Logical Specification. In order to construct models of the real world certain groups of observations are selected from the infinity of observations that can be made. Those models are then constrained to behave in a particular way so that one can know all about them. It is not necessary to know what happens in reality in order to know what happens in the model: once constructed the model can be divorced from reality.

If we look at the definitive script we see that the modelling is *not* independent of the context. Indeed we can say it is *situated modelling*. The modelling is constantly being situated in its real world context as it is being formulated. In order to have a realistic model of what I observe, any incremental change in my model must be consistent with what I observe, otherwise I need to change the model to make it

so. A script in that sense is tentative, as our beliefs about the world are. We believe that the world behaves thus because of experience, but we are ready to amend our beliefs should the experience subsequently require.

We can contrast this idea of immediacy by comparing design with analysis. Analysis has to do with understanding what is, whereas design is to do with what might be. Synthesis is to design what analysis is to research. Conventional computational approaches suit analysis, differentiating and discriminating what is. Definitive methods are to do with integrating and generalising - more typical of design processes.

These ideas do not appear to be as explicit in non-definitive systems. For example it is perhaps an open question whether these notions are apparent (or could be constructed) in such programming paradigms as APT, PADL-2 and application programs like Design View. On the other hand the power of these methods and the growing developments of the parametric approach to CAD systems generally bear witness to the perceived need for more "situated modelling" techniques.

10.32 Other Definitive Methods

EdenLisp derives from other definitive notations based on Eden, but it is significantly different in its conception and implementation. Unlike Eden, EdenLisp emphasises the definitive nature of the code by deliberately separating the delineation of functions from the script of definitions that call those functions. In that respect it has more in common with derivatives of Eden such as DoNaLD and CADNO. However, those notations translate to Eden whereas EdenLisp is the basic notation. In Eden, procedures and functions tend to be the principal elements of scripts in terms of the bulk of the programming effort, so the definitive nature of what is being coded is difficult to see. Indeed some of the code written by others in Eden looks more like C-program code. By contrast, the process of constructing functions separately in EdenLisp has proved a strong discipline. Since all functions have to be typed and declared, much thought has to go into their construction. The user is thus encouraged to create functions that are as generic and useful as possible so they can be added to libraries for use by other scripts. Examples of such functions abound in the function libraries of EdenLisp such as EdenUtil, EdenTop, EdenGeom, and EdenDisp. Those contain functions that are respectively general utilities such as sorting, replacing items in a list, set operations, topological and geometrical relationships and finally display calls to AutoCAD. Many functions are structured so as to make it easy to call another function, *e.g.* display functions

always need a call to the AutoCAD "command" functions, so the name of the command can be passed via a string, using various pseudonyms to enable access to different ways of constructing particular shapes.

It might be argued that the use of AutoLisp rather than EdenLisp to generate EdenLisp functions is "impure" since Eden functions are generated by Eden statements. That is true and some guarding is done by imposing the requirement to declare and type the functions. (Eden does allow calls to C functions. DoNaLD too has the ability to call Eden directly rather in the manner that one might include Assembler code "in line").

10.4 The Future

10.41 User interface

The Engineering designer wants computer tools that will do the job without having to worry about how they operate, and preferably that they do it better and faster than anything else. From that point of view objections can be raised concerning the practicality of definitive methods. The user interface is textual. It requires a sophisticated programmer with a good understanding of the definitive method to make any headway with the system. Typically a new user needs prior knowledge of a language like C to work with Eden, and to use EdenLisp the user needs both to be able to use AutoCAD reasonably well, and to be able to program in AutoLisp at the level expected of a good programmer in Lisp.

Those issues are important, but it was thought that definitive principles themselves needed to be clarified and implemented before considering the user interface in detail. Some work has been done on definitive user interfaces. A number of experiments by other workers in the research group have shown the feasibility of using text windows and graphical buttons and so on in a graphically based definitive system, *e.g.* SCOUT, [Beynon, Yung & Hogan, 1992]. It would not be difficult to write a similar interface using EdenLisp and AutoCAD facilities that will itself be definitive in form. That would mean that the display and the means of interaction would itself be a function of that which is being designed. Different views of the design demand different interfaces so it makes sense to work in that direction. Although it is difficult to imagine a totally non-textual interface for EdenLisp in the formation of definitions, one can envisage many short-cuts that can be used in creating geometrical forms without the need for much typing. The objections relating to the user interface are not therefore insuperable. Whilst not trying to defend the current situation, one can perhaps reflect that many interface

languages to popular software are also rather obscure. Most CAD systems have macro or language facilities that defy easy entry. The popular Matlab provides another example of the difficulty in making specifications.

10.42 Actions

Actions are crucial to the definitive system. Without them the system is only a "graphical spreadsheet". Whilst an extremely useful tool, it would be a mere extension of parametrics, somewhat in the manner of Design View. The significance of actions is that they cause redefinition indirectly. An incremental change in a script is easy to do by entering a new definition in place of an existing one, but to make a massive change that might be required to achieve a desired state might require a significant amount of redefinition, much of it of a tedious and repetitive character. The role of an action is to carry out redefinition, activated by means of a trigger (the guard in a concurrent system). The trigger is usually a variable that when changed by a redefinition input by the user (or an agent) causes the action to take place. Thus a single change of variable might be responsible for setting off wholesale changes in a script; indeed it might even cause groups of scripts to change! Such changes are obviously dangerous. The constraints explicitly set by an agent are under the control of that agent, but actions unwittingly set off by triggers of which the agent is unaware could cause chaos. These are the problems of what are called autonomous agents: those set up by the computer to do things like change the screen layout. Clearly much will need to be done to keep tabs on such autonomous actions!

In EdenLisp the approach taken to that problem of actions is to make the triggers explicit. Redefinition is actioned by calling specific EdenLisp functions. The user is therefore always aware of the action being taken. However it is not at all clear at this stage whether that will continue to be the case when concurrency is implemented. There will have to be some further work to deal with that.

10.43 Multi-Agent systems and Concurrency

The problems of the windowing or agent oriented environments discussed in chapter 6 need to be addressed in order to develop the method for multi-agent scenarios. In the paper [Adzhiev, Beynon, Cartwright, Yung, 1994*a*] the issues that are identified relate to the development of the Abstract Definitive Machine (ADM): to the ways that agents must interact, how to structure the kinds of changes that are permitted within and between scripts so as to allow for the following.

- Parallel redefinition among a group of scripts to permit several agent actions to occur simultaneously,
- Automatic detection of conflicts, such as when two agents attempt to redefine the same variable the ADM prompts the user to resolve the situation.
- Exceptional privilege of super user, allowing interaction to direct the computation as an independent and unspecified agent within the ADM.

In practice the structure will need to be very complex. Agents may work in several different modes: they may be free to develop their own designs independently or one may have a leading role and constrain others.

It is the responsibility of a Design Co-ordinator as Super-User to make decisions at all levels. Milestones in a design would then correspond to high level decision points, by analogy with a design conference of all agents operating at that stage. In principle different sets of agents can be involved at different milestones. After such decisions, the specification becomes fixed and all affected variables inherit that information.

Between milestones, agents would be autonomous; they could experiment with their own scripts. Those independent experiments, corresponding to a kind of design folio on the part of the experimenters, must eventually yield a proposal for the modification of the virtual prototype that furthers the actual design. The Co-ordinator's Role in this experimental stage would be to

- direct patterns of communication to settle ways that agents relate common variables to one another between the milestones (e.g. what material to use could vary)
- identify what is to be fixed, establishing goals at milestones that select from the variant scripts generated in each agent's design folio.
- estimate progress and degree of consensus
- impose constraints and schedules to build up the components so as to synthesise the design in a "bottom up" manner, and hence progress the design of the product.

It is hoped that the development of a multi-agent system would realise the dream of a truly concurrent design approach, providing a single unifying model (the "virtual prototype") for investigating many views, and using computing metaphors that are consistent between different modelling needs.

10.5 Conclusions

1. *Engineering Design is based upon observation and experiment. Conception is associated with heuristics, abstracting on the basis of incremental behaviour. Invention appears to require the ability to accept the common experience of others only provisionally, being prepared to make new relationships that lead to novel designs.*

2. *The servicing of conceptual design has been done historically by experiment on prototypes - typically physical models that capture information in such a way that new observations can be made.*

3. *Computational models are symbolically based and therefore have a form that cannot capture content except in a preconceived way. Symbols have the power to suggest content well beyond their internal application and that is the basis of much computer aided design. The user may be provided with an empty space but with pre-conceived, but suggestive, symbols around to stimulate and support.*

4. *The computational method based on definitive methods provides that empty space in the form of latent states, where the forms and symbols are invested with significance by the user on the basis of experiment, rather than being pre-conceived.*

5. *The state of a definitive script is that of the incomplete exploration of possible behaviours and relationships. Incremental changes in the script are metaphors of physical behaviour. A set of scripts can be thought of as a virtual prototype with similar properties to a physical one: experiments can be performed, autonomous agents can act concurrently on the prototype, new information may be extracted by "what if ...?" explorations.*

6. *The virtual prototype leads to a single unifying model for investigating many views, with metaphors that are consistent between different modelling needs*

7. *EdenLisp represents a definitive notation that captures the ideas described above. It illustrates in principle the feasibility of virtual*

prototyping by a single user; it has the potential to develop a multi-agent approach via the windowing environment.

8. *Actions in EdenLisp enable definitive scripts to be created and edited. They can cause incremental or massive changes in scripts that reflect substantial but predictable behaviour in the object world. New objects can be created, new relationships formed and different modelling environments can be called. The power of actions is immense and checking is likely to be a significant area for future work.*

9. *EdenLisp shows that it is possible to link definitive notations to commercial and other programming approaches in a synergistic way: the designer is aided with routine work whilst freed to carry out experiments of a conceptual kind.*

10. *The experimental approach is educationally interesting. It deals with both discrimination and generalisation of declarative material, cognitive processes that are the easiest stages of learning and which stimulates procedural thinking. In the EdenLisp environment the student of design is encouraged to think in a generic way, leading to creative and fruitful products.*

List of References

Abbreviation in the text	*Reference Details*	*Section where quoted*
Abelson & Sussman, 1985	Abelson H & Sussman GJ, *'Structure and Interpretation of Computer Programs'*, MIT and McGraw Hill, 1985	3.23 5.13 6.1
Adzhiev, Beynon, Cartwright, Yung, 1994*a*	Adzhiev V, Beynon WM, Cartwright AJ, Yung YP, *A* Computational Model for Multi-Agent Interaction in Concurrent Engineering' *Proc. 2nd Int Conf. Concurrent Engg and Electronic Design Automation. Poole. April 1994*	10.3
Adzhiev, Beynon, Cartwright, Yung, 1994*b*	Adzhiev V, Beynon WM, Cartwright AJ, Yung YP, 'A New Computer-based Tool for Conceptual Design', *Int.Wksh. Comp.Aided Conceptual Design Lancaster. April 1994*	10.3
Akman & ten Hagen, 1989	Akman V & ten Hagen PJW: 'The Power of Physical Representations', *Intelligent CAD Systems III. Eds V Akman, PJW ten Hagen, PJ Veerkemp. Springer Verlag 1989. 126-145.*	3.12
Autodesk, 1992	*AutoLisp Release 12 Programmer's Reference.* Autodesk Inc., 1992	6.1
Autodesk, 1992	*AutoCAD Release 12 Customisation Manual*, Autodesk Inc. 1992	9.32
Backus, 1978	Backus JW, 'Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs', *Communications of the ACM, Vol 21, August 1978, 613-641*	3.13
Beynon, 1986	Beynon WM, 'ARCA - A Notation for displaying and manipulating combinatorial Diagrams'. *U. of Warwick. Comp Sc. RR#76. 1986*	3.41
Beynon, Angier, Bissell & Hunt, 1986	Beynon WM, Angier D, Bissell T, Hunt S; 'DoNaLD, a Line Drawing System based on Definitive Principles'; *U.of Warwick Comp.Sc RR#87, 1986*	3.41
Beynon, 1987	Beynon WM, 'Definitive Principles for Interactive Graphics', *NATO ASI Series F:40 1987. 1083-1097*	3.41

Beynon & Yung, 1988	Beynon WM &Yung E, 'Implementing a Definitive Notation for Interactive Graphics', *Proc CG '88*	4.22
Beynon & Cartwright, 1989	Beynon WM & Cartwright AJ, 'A Definitive Approach to the Implementation of CAD Software', *Intelligent CAD Systems III. Eds V Akman. PJW ten Hagen. PJ Veerkemp. Springer Verlag 1989, 126-145*	3.42 4.21
Beynon, 1990	Beynon WM, 'Parallelism in a Definitive Programming Framework', *Proc Conf on Parallel Computing. Elsevier Sc. Pub. 1990. 425-430*	1.2 10.1 2
Beynon *et al.* 1990	Beynon WM, Norris MT, Orr RA, Slade MD, 'Definitive Specification of Concurrent Systems', *Proc.UKIT'90. IEE Conf.Publications 316, 1990 52-57*	5.13
Beynon, 1992	Beynon WM, 'New paths for programming in theory and practice' *Seminar at IBM Warwick Software Development Lab. Sept 1992*	3.1 4.2
Beynon, Cartwright, Yung & Hogan, 1992	Beynon WM, Cartwright AJ, Yung YP, Hogan PJ, 'Scientific Visualisation Exper-iments and Observations', *Proc 3rd Eurographics Workshop on Visualisation in Scientific Computing. Viareggio, 1992, 157-173.*	5.13
Beynon & Cartwright, 1993	Beynon WM & Cartwright AJ, 'Agent-oriented Modelling for Engineering Design', *Proc. Int. Conf. on Computer-aided Design. Yalta. Russia. May 1993.*	5.13
Beynon, Joy, Cartwright & Godfrey, 1993	Beynon WM, Joy M, Cartwright AJ, Godfrey KR, 'Agent-oriented Modelling for Interactive Systems', *SERC grant. 3-year Research Fellowship for S.Yung from May 1993*	8.11
Beynon, *et al*, 1993	Beynon WM, Norris MT, Russ SB, Slade MD, Yung, YP, Yung YW, 'Software Development as Agent-oriented Modelling: an Illustrative Example.' *In preparation*	5.13
Brown, 1982	Brown CM, 'PADL-2: A Technical summary' *IEEE Computer Graphics & Applications V2 N2. Mar 1982. 69-84*	4.12
Burns & Stalker, 1986	Burns T & Stalker GM "The Social Context of Innovation" *in "Product Design and Technological Innovation". Ed Ray & Wield. Open University Press. 1986.*	9.1
Butchard & Cartwright, 1993	Butchard, Helen, 'Commissioning of PADL-2 - a Developmental Solid Modelling Package', *3rd Year B.Eng. Project. supervised by AJ Cartwright. U. of Warwick. Dept of Engg.* 1993	1.12
Cartwright & Beynon, 1992	Cartwright AJ & Beynon WM, 'Enhancing Interaction in Computer-aided Conceptual Design', *Proc Int Conf on Manufacturing Automation. Hong Kong. Aug 1992. 643-648*	2.5 4.44

Case, 1992	Case K, 'Feature Technology - an Integration Methodology for CAD & CAM', *Proc Int Conf on Manufacturing Automation. Hong Kong, Aug.1992. 613-624*	5.12
Chan & Paulson, 1987	Chan WT & Paulson BC Jr, 'Exploratory Design Using Constraints AI for Engg Des', *Anal & Manuf. V1.No1 Academic Press 1987*	2.3
Clocksin & Mellish, 1987	Clocksin WF & Mellish CS, '*Programming in Prolog*', 3rd Edn., Springer Verlag, 1987	3.21
Coplin, 1989	Coplin JF, "Engineering design - a powerful influence on the business success of manufacturing industry", *Proc IMechE Int Conf Engineering Design. Harrogate 1989, 1-31*	2.1
Cross, 1989	Cross, N. "*Engineering Design Methods*", J Wiley, 1989	9.1
Davies, 1992	Davies P, *The Mind of God*, Penguin books, 1992	2.1
DesignView Manual, 1991	'DesignView Version 3 for Windows', *Program documentation. ComputerVision CADDS DesignView Group. 1991*	4.14
Dym, 1987	Dym CL, 'Issues in the design and implementation of Expert Systems', *Artificial Intelligence for Engineering Design. Analysis & Manufacturing. Vol1. No1. 1987. 37-46*	3.31
Ehrlenspiel & Dylla, 1989	Ehrlenspiel K & Dylla N, 'Experimental Investigation of the Design Process', *Proc IMechE Int Conf Engineering Design. Harrogate 1989. 77-95*	1.1
El Dahshan & Barthes, 1989	El Dahshan K & Barthes JB, 'Implementing Constraint Propagation in Mechanical CAD Systems'. *Intelligent CAD Systems III. Eds V. Akman. PJW ten Hagen. PJ Veerkemp. Springer Verlag 1989 217-227.*	2.4
Foley	Foley J, 'Models & Tools for the Designers of Computer User-Interfaces', *NATO ASI Series F. Computer and Systems Sciences. Vol 40. 1121-1152*	3.42
French, 1985	French M, *Conceptual Design for Engineers*. Design Council, Springer Verlag, 2/e 1985	2.1
Glaser, Hankin & Till, 1984	Glaser H, Hankin C, Till D. '*Principles of Functional Programming*'. Prentice Hall. 1984	3.13
Goldberg & Robson, 1983	Goldberg A & Robson, D '*Smalltalk-80. The Language and its Implementation*'. Addison Wesley, May 1983	3.2
Gu, ElMaraghy & Hamid, 1989	Gu PH, ElMaraghy HA, Hamid L. 'FDDL: A feature based design description language'. *Design Theory & Methodology. ASME 1989*	2.2
Hartquist & Marisa, 1983	Hartquist EE, Marisa HA, 'PADL-2 Users' Manual', *Cornel Programmable Automation. Ithica. NY. 1983*	4.1

Hofstadter, 1979	Hofstadter DR, *'Godel, Esher, Bach: An Eternal Golden Braid'*, Penguin Books, 1979	2.2
Jones, Maynard & Stewart, 1990	Jones R, Maynard C, Stewart I, *'The Art of Lisp Programming'*, Springer Verlag 1990	6.1
Kelly, *et al.*, 1986	Kelly P, Kranzberg M, Rossini F, Baker N, Tarplay F, Mitzner M 'Introducing Innovation', *in "Product Design and Technological Innovation", Ed Ray & Wield, Open University Press, 1986.*	9.1
Koren, 1983	Koren, *'Computer Control of Manufacturing Systems'*, McGraw Hill, 1983	4.11
Koegal, 1989	Koegal JF, 'Planning and Explaining with Interacting Expert Systems' *Intelligent CAD Systems III, Eds V.Akman, P.J.W ten Hagen, P.J.Veerkemp, Springer Verlag 1989, 17-31.*	3.32
Krause, Vosgerau, & Yaramanoglu, 1989	Krause F, Vosgerau FH, Yaramanoglu N 'Implementation of Technical Rules in a Feature-based Modeller'. *Intelligent CAD Systems III, Eds V.Akman, P.J.W ten Hagen, P.J.Veerkemp, Springer Verlag 1989, 195-208.*	2.2 3.32
Leyton, 1988	Leyton M, 'A Process Grammar for Shape', *Artificial Intelligence, V34, No2, March 1988, 213-248*	3.31
McKay, 1988	McKay A, 'The Structure Editor Approach to Product Description', *Information Support Systems for Design and Manufacture, PDS Report, University of Leeds, 1988.*	3.31
Mäntylä, 1990	Mäntylä M, 'A Modeling System for top-down design of assembled products'. *IBM J.Res. Develop. V.34. No5. Sept 1990*	2.2
Michie, 1986	Mitchie D, *'On Machine Intelligence'*, 2nd Edition, Ellis Horwood, 1986, ch 16, 18	2.2, 3.32
Miller, *et al*, 1980	Miller GM, Kuchar NR, Douglas RJ, 'CAD/CAM in Primary Manufacturing Processes' *The CAD/CAM Handbook, ComputerVision Corporation ,1980, chapter 5.9.*	3.11
Miller, 1989	Miller W, 'Implementation of CAD in W Germany' *Seminar at Univ. of Warwick Business School, Oct 1989*	8.1
Minsky, 1981	Minsky, Marvin: 'A Framework for Representing Knowledge', *in 'Mind Design, Philosophy, Psychology and Artificial Intelligence', Ed. Haugeland J. MIT Press, 1981, 95-128*	2.2
Oh, Langdown, Sharpe, 1994	Oh V, Langdown P, Sharpe JEE, 'Schemebuilder: An Integrated Computer Environment for Product Design ', *Proc Int'l Wksh on Computer Aided Conceptual Design, U of Lancaster, April* 1994	5.13
Piela, *et al*, 1992	Piela P, Katzenberg B, McKelvey, 'Integrating the User into Research on Engineering Design Systems' *Res. Engineering Design (1992) 3:211-221*	1.1

Plato	Plato, *Republic*, Peguin edition, 1955, VII, 6	2.4
Popplestone, 1984	Popplestone RJ; 'The Application of AI Techniques to Design Systems'; *Int Symp on Design and Synthesis, Japan Soc of Prec.Engg Tokyo 1984*	5.12
Popplestone, *et al.* 1986	Popplestone RJ, Smithers T, Corney J, Koutsou A, Millington K, Sahar G. 'Engineering Design Support Systems'. *IKBS/MS 7/86 3.1*	5.12
Pugh & Morley, 1988	Pugh S & Morley IE 'Total Design, Some Questions and Some Answers', *Design Division, Univ of Strathclyde*, 1988	2.3
Pye, 1983	Pye D, '*The Nature and Aesthetics of Design*', Herbert Press, 1983	1.1
Ritchie & Thompson, 1984	Ritchie G, Thompson H, 'Natural Language Parsing', in '*Artificial Intelligence - Tools, Techniques and Applications*'. *Eds. O'Shea T. & Eisenstadt M., Harper & Row, 1984 ch.11*	2.2
Rooney & Steadman, 1987	Rooney J & Steadman P, '*Principles of Computer Aided Design*'. Chap.1, Pitman Press 1987	4.22
Roth, 1981	Roth KH, 'Foundation of Methodical Procedures in Design'. *Design Studies V1,No2,April 1981, 107-115*	1.1
Roth, 1989	Roth KH, 'Methods and Relationships for automatic design of connections by the computer', *Proc IMechE Int Conf Engineering Design, Harrogate 1989, 637-654.*	2.1
Rossignac, Borel & Nackman 1989	Rossignac JR, Borel P, Nackman LR, 'Interactive Design with Sequences of Parameterized Transforms', *Intelligent CAD Systems III, Eds V.Akman, P.J.W.ten Hagen, P.J.Veerkemp, Springer Verlag 1989, 93-125.*	5.12
Sandström, 1968	Sandström CI, '*The Psychology of Childhood and Adolescence*' Penguin Books, 1968	2.2
Siddall, 1982	Siddall James N. *Optimal Engineering Design* Dekker, 1982.	9.2
Shaw, Bloor & de Pennington, 1989	Shaw NK, Bloor M Susan, de Pennington A, 'Product Data Models', *Research in Engineering Design, Vol1, No1, 1989, 43-50*	3.31
Smith BC, 1987	Smith, Brian Cantwell: 'Two Lessons of Logic', *Computer Intelligence, V3, 1987, 214-218*	2.1
Smith ST & Chetwynd, 1992	Smith ST & Chetwynd DG, *Foundations of Ultraprecision Mechanism Design*, Gordon & Breach Sc Pub, 1992, ch.4	7.3
Smithers, 1987	Smithers T: 'AI-based Design v Geometry based Design Workshop on AI in Civil Engineering', *AI Applications Inst Edinburgh, Nov 1987*	4.22

Sobolewski, 1988	Sobolewski M, 'Percept Conceptualisations and their Knowledge Representaion Schemes', *Proc 6th Int.Symp. on Methodologies for Intelligent Systems, Oct*1988, *236-245*	2.2
Spillers & Newsome, 1989	Spillers WR & Newsome S, 'Design Theory: A Model for Conceptual Design', *Proc. 1988 NSF Grantee Workshop on Design Theory and Mehodology, Springer Verlag, 1989*	2.3
Starkey, 1992	Starkey CV, *Engineering Design Decisions*, Edward Arnold, 1992	1.1, 9.1
Stidwell, 1989	Stidwell MJ, The CADNO Programming Language, *U. of Warwick Comp. Sc. Report, May 1989*	5.21
Sudkamp, 1988	Sudkamp TA, '*Languages and Machines*' Addison Wesley,1988	3.2
Sutherland, 1963	Sutherland LE, "SKETCHPAD: A Man-Machine Graphical Communication System, *TR#296, Lincoln Laboratory, MIT, 1963*	2.4
Takala, 1989	Takala T, 'Design Transactions and Retrospective Planning', *Intelligent CAD Systems III, Eds V.Akman, P.J.W.ten Hagen, P.J.Veerkemp, Springer Verlag 1989, 262-272*	2.4
Tan, *et al*, 1987	Tan ST, Yuen MF, Sze WS, Yung.WY: 'NC Machining Algorithms for CSG Solids'. *Advances in Design Automation, Vol.1, ASME Design Automation Conf., Boston Ma. Sept 27, 1987*	4.12
Tanimoto, 1990	Tanimoto, SL *The Elements of Artificial Intelligence*, Computer Science Press 1990	6.2
Tomiyama & ten Hagen, 1987	Tomiyama T & ten Hagen PJW, 'The Concept of Intelligent Integrated Interactive CAD systems', *CWI report, Centre for Mathematics & Computer Science, Amsterdam, 1987*	6.4
Tomiyama, 1989*a*	Tomyama T, 'Meta-model: A key to Intelligent CAD Systems', *Research in Engineering Design, Vol1, No1, 1989, 19-34*	3.1
Tomiyama, 1989*b*	Tomyama T, 'Object Oriented Programming Paradigm for Intelligent CAD Systems', *Intelligent CAD Systems III, Eds V.Akman, P.J.W.ten Hagen, P.J.Veerkemp, Springer Verlag 1989, 3-16.*	3.1
Tomiyama, *et al.* 1994	Tomiyama T, Kiriyama T, Umeda Y, '*Towards Knowledge Intensive Engineering*', Proc Int Wksh Computer Aided Conceptual Design, Univ of Lancaster, April 1994	5.13
Trigg, 1973	Trigg R, *'Reason and Commitment'*. Cambridge Univ. Press, 1973	2.2
Turner, 1987	Turner D, 'An Overview of Miranda', *Bull EATCS, 103-114*. 1987	3.22
Ullman, 1992	Ullman DG, *'The Mechanical Design Process'*, McGraw Hill, 1992	2.1, 9.1

van Houten & van t'Erve, 1992	van Houten FJAM & van t'Erve AH, 'Feature Based Manufacting with PART', *Proc Int Conf on Manufacturing Automation, Hong Kong, Aug.1992, 464-475*	5.12
Veerkamp, 1992	Veerkamp P 'On the Development of an Artifact and Design Description Language', *CWI, Amsterdam, 1992*	2.3 5.13
Voss, Winch, 1989	Voss C, Winch G. 'Organisational Linkages for CAM/CAM Implementations', *Seminar presented U. of Warwick, Oct.1989*	8.1
Wærn, 1986	Wærn Y, '*Cognitive Aspects of Computer Supported Tasks*', J Wiley, 1986	2.2 5.11 9.1
Whitney, 1992	Whitney DE, 'Japanese CAD Methodologies for Mechanical Products', *IMechE/SERC Seminar, 21 Sept,* 1992	2.5
Wilson & Greaves, 1989	Wilson PM, Greaves J, 'Forward Engineering - A Strategic Link between Design and Profit'. *Mechatronics Conference, Lancaster, Sept.1989*	2.5
Wilson, 1972	Wilson RJ: *Introduction to Graph Theory*, Oliver & Boyd, 1972	4.22
Winston & Horn, 1989	Winston PH & Horn BKP, *LISP*, 3rd Edn, Addison Wesley, 1989	6.1
Woo, 1985	Woo TC 'A Combinatorial Analysis of Boundary Data Structure Schemata ', *IEEE Comp Gr and App. March 1985*	4.22
Yung, 1987	Yung EYW, 'EDEN - Evaluator of DEfinitive Notations', *UG Project Report, Dept of Comp.Sc. U.of Warwick, 1987*	1.2, 6.1

Appendix A

Formal Language Definitions for EdenLisp

1. Productions

Lisp implementation of a recursive descent parser for the following grammar

```
1. program          ::=   statement
2. statement        ::=   identifier = expression
                        | identifier = if relational_expr then
statement
                                                          else
statement
3. expression       ::=   term opr1 expr
                        | term
4. relational_expr  ::=   "=" | "<=" | ">=" | ">" | "<" | "/="
5. term             ::=   factor opr2 term
                        | factor
6. factor           ::=    unsigned const (id | int | real)
                        |  signed const   (int | real)
                        |  signed variable (id)
                        |  "(" expression ")"
                        |  function "(" factor ")"
                        |  set constructor
7. opr1             ::=   "+" | "-"
8. opr2             ::=   "*" | "/"
9. int              ::=   0|1|2|3|4|5|6|7|8|9
10. real            ::=   int* "." int* |int* "." |"." int*
11. id              ::=   a..z*[A..Z]*[int]*[_$£#]
12. function        ::=   basic Autolisp function
                        | Valid EdenLisp function
                        | AutoCAD commnand
13. set             ::=   "[" expression "]"
```

2. Valid Operators in EdenLisp

Algebraic operators
+ - * / 1- 1+ ~ ABS EXP EXPT LOG REM SQRT ATAN COS GCD LSH MAX MIN
SIN ASCII ATOI ATOF ANGTOS CHR FIX FLOAT ITOA LENGTH EVALID
PROJN OBJECT POINT VSUM VDIFF VTRANS VSHEAR VSCALE WRITE LOCATE
> >= /= = <= < IF THEN ELSE EQ EQUAL AND BOOLE LOGIOR LOGAND
NOT NULL OR ATOM BOUNDP LISTP MINUSP NUMBERP ZEROP
SIMPLEX COMPLEX MORPH STOL PEDGE CEDGE CGRAPH FOREST SHAPE ISOMORPH EXTRUDE ROTOBJ
STRLEN STRCAT STRCASE SUBSTR ANGLE DISTANCE INTERS POLAR TRANS

Geometical/Draw functions
WIREFRAME SURFACE LINE PLINE _3DPOLY ARC CIRCLE SPLINE TXTWIN LABEL
GRREAD GRTEXT GRDRAW GRCLEAR VPORTS TRANS HANDENT HINGEH hingev
ENTUPD ENTMOD ENTSEL ENTLAST ENTNEXT ENTDEL ENTGET OSNAP REDRAW GRAPHSCR

AutoLisp commands
ERROR _VER ALLOC APPEND APPLY ASSOC CAAAAR CAAADR CAAAR CAADAR CAADDR CAADR CAAR CADAAR CADADR CADAR CADDAR CADDDR CADDR CADR CAR CDAAAR CDAADR CDAAR CDADAR CDADDR CDADR CDAR CDDAAR CDDADR CDDAR CDDDAR CDDDDR CDDDR CDDR CDR CLOSE COMMAND COND COND CONS DEFUN ENAME EVAL EXIT EXPAND FINDFILE FOREACH FUNCTION GETANGLE GETCORNER GETDIST GETINT GETKWORD GETORIENT GETPOINT GETREAL GETSTRING GETVAR INITGET LAMBDA LAST LIST LOAD MAPCAR MEM MEMBER MENUCMD NTH OPEN PAGETB PAUSE PICKSET PRIN1 PRINC PRINT PROGN PROMPT QUIT QUOTE READ-CHAR READ-LINE REPEAT REVERSE SET SETQ SETVAR SSADD SSDEL SSGET SSLENGTH SSMEMB SSNAME SUBST TBLNEXT TBLSEARCH TERPRI TRACE TYPE UNTRACE VER VMON WHILE WRITE-CHAR WRITE-LINE

Type words
INT REAL STR Lreal POINT Lstr LLstr LLreal LIST SYM FILE ENAME WIN FRAME LOCAL

3. Environment for each type or type group

1. type group that is input to the operation
2. list of ops and the return type after the op is performed

INT (+ . int) (* . int) (- . int) (/ . int) (1- . int) (1+ . int) (ABS . int) (REM . int) (GCD . int) (MAX . int) (MIN . int) (FLOAT . real) (CHR . str) (ITOA . str) (LIST . Lreal) (POINT . Lreal) (EVALID . int)

REAL (+ . real) (- . real) (* . real) (/ . real) (1- . real) (1+ . real) (REM . real) (ABS . real) (EXP . real) (LOG . real) (EXPT . real) (SIN . real) (ATAN . real) (COS . real) (SQRT . real) (GCD . real) (MAX . real) (MIN . real) (expt . int) (FIX . int) (RTOS . str) (ANGTOS . str) (LIST . Lreal) (POINT . Lreal) (EVALID . real)

STR (LIST . Lstr) (PEDGE . LLstr) (STRCASE . str) (STRCAT . str) (STOL . Lstr) (ASCII . int) (ATOI . int) (STRLEN . int) (ATOF . real) (EVALID . str)

LREAL (LIST . LLreal) (PLUS . Lreal) (pt- . Lreal) (POINT . Lreal) (INTERS . Lreal) (LENGTH . int) (DISTANCE . real) (ANGLE . real) (Vsum . Lreal) (Vdiff . Lreal) (LOCATE . Lreal) (Vtrans . Lreal) (Vshear . Lreal) (EVALID . Lreal) (PROJN . real)

LLREAL (Vsum . Lreal) (Vdiff . Lreal) (PEDGE . LLreal) (CEDGE . LLreal) (CGRAPH . LLreal) (FOREST . LLreal) (EVALID . LLreal)

```
LSTR (LIST . LLstr) (PEDGE . LLstr) (CEDGE . LLstr) (CGRAPH . LLstr)
     (FOREST . LLstr) (EXTRUDE . LLstr) (SHAPE . LLstr) (STRCAT . STR)
     (MKPOLY . LLstr) (MORPH . LLstr) (EVALID . Lstr)

INT LREAL          (PROJN . real)
INT LLREAL         (PROJN . Lreal)

STR LLSTR          (ISOMORPH . LLstr)
STR INT            (SUBSTR . str)

LLSTR              (COMPLEX . LLreal)

LREAL REAL         (POLAR . Lreal) (VScale . Lreal)
LREAL REAL STR     (TXTWIN . str) (LABEL . str)
LREAL LLREAL       (PROJN . Lreal)

LLREAL REAL        (VSCALE . LLreal) (ROTOBJ . LLreal)
LLREAL LREAL       (VSHEAR . LLreal) (VTRANS . LLreal) (OBJECT . LLreal)
LLREAL STR         (WIREFRAME . LLreal)
```

Appendix B

EdenLisp Scripts

The following Scripts are written in the definitive notation EdenLisp.

Script 1. Tumbler Mixer Machine: Shaft Design
Calculations for the shaft size to carry the steady and impact loading on a tumbler mixer machine with a payload of four tonnes of powder of density about 1000 kg/m^3

```
; Tumbler Mixer Machine
; by AJ Cartwright
; August 1993

real : pi R1 R2   L2 L1   Ws Wp Wf Wm   ; all terms defined as used
real : F1 F2 K    Mm Ma Tm Ta
real : Oy Oe Os   D Do    RF Me Te Trt

Wf = 15000.0                ; Frame + flask weight (N)
Wp = 28000.0                ; Powder weight (N)
Ws = Wf + Wp                ; Static weight = 43000(N)
L1 = 2484.0                 ; Bearing spacing (mm)

; Shock load                ; Assume 2/9 of powder shifts vertically by 1200mm
F1= sqrt(4*Wp*1200*K/9)     ; F1 = powder surface & flask vertical on fall = 223 109 N
F = F1/2.0                  ; F2 = second case: flask horizontal = 111 554 N
K = 40000.0/12.0            ; K  = stiffness of the shaft/flask system. estimated by a
                            ;  deflection of 12mm under 4 tonnes load  = 3333.3 N/mm
R1 = Wm/2.0                 ; Bearing reaction R1 = 44316 N
R2 = -Wm/2.0                ; bearing reaction R2 = -44316 N


; Moments and torques
Mm = Wm * L1/10.0           ; BM = (Ws + mean shock) x L2 = 2.20162 e7 N mm
Ma = F1 * L1/10.0           ; variable shock BM = 20 000 x L2 = 5.54203 e7 N mm
```

```
Tm = 5 000 000          ; mean torque = 5,000 N.m
Ta = 250.0*F1           ; variable shock torque = 55 777 250 N.mm
                        ; mean torque/rev using bearing formula Wm = 21 453 Nm
                        ; Horizontal reaction on tie rod 0.5m from c/l = 42 906 N

; Min Shaft Diameter
Oy = 600.0              ; yield strength of shaft material (MPa)
Oe = 400.0              ; endurance strength (MPa)
RF = 2.0                ; reserve factor
Te = Tm/Oy + Ta/Oe      ; effective torque (N mm)
Me = Mm/Oy + Ma/Oe      ; effective bending moment (N mm)
Trt= sqrt(Te^2+Me^2)    ; Equivalent rotating-bending load
d  =(32*RF*Trt/pi)^0.33 ; Minimum shaft diameter = 167.149 mm
De = 1.08*Do            ; Hollow shaft with 25 mm wall thickness = 180.531mm
```

Script 2, 4-Bar Linkage

Design for 4-bar linkage with animation by varying the angle theta, between the driver, bar A, and the horizontal

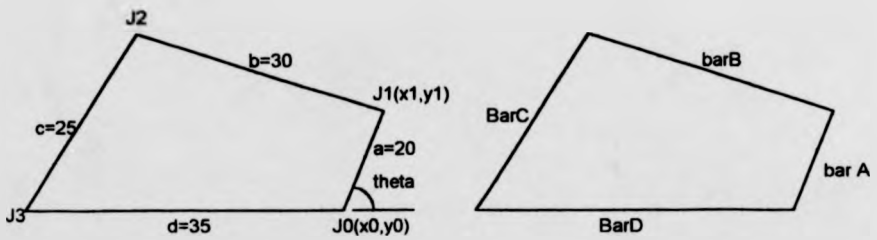


Fig B1.1 Notation and display for 4-bar linkage

```
; 4-bar linkage for EDENLISP
; by A.J.Cartwright
;
; Date: August 1991
; mods. 13 Dec 1992,
;       29 Mar 1993

 Lstr : nodes1                    ; list of strings for end labels of bars
LLstr : nodes2                    ; describes connections of labels
 real : x y a b c d L M K S sq    ; declare variables (defined as used)
 real : theta delta
Lreal : J0 J1 J2 J3 J4 J5 J6 J7   ; points for ends of 4-bar. label positions

nodes1= ["J0", "J1", "J2", "J3"]  ; labels for end points
nodes2= cedge(nodes1)             ; see text below

theta = 1.0                       ; angle of driver (rad)
```

```
    a = 20.0                              ; length of driver arm (mm)
    b = 30.0                              ; length of cross member (mm)
    c = 25.0                              ; length of driven arm  (mm)
    d = 35.0                              ; distance between fixed pivots of a and c

    L = - sin(theta)                      ; computations to establish angle delta
    M = (d/a) + cos(theta)
    S = a*a + c*c + d*d - b*b
    K = d*cos(theta)/c + S/(2*a*c)
   sq = sqrt (L*L + M*M - K*K)
delta = 2*atan((L + sq)/(M - K))          ; compute angle delta from given data

    x = -(d + c*cos(delta2))              ;  hence find co-ords of point J2
    y = c*sin(delta2)

   J0 = [0.0,0.0]                         ; coords of fixed pivot of driver bar a
   J1 = [a*cos(theta), a*sin(theta)]      ;  coords of  moving end of driver bar a
   J2 = [x, y]                            ;  coords of  driven end of bar c
   J3 = [-d,0.0]                          ;  coords of  fixed pivot of driven bar c

frame : barb barf bard                    ; declare object variables

 barb = complex (nodes2)                  ; create variables making up the 4-bar
 barf = object(barb,[300,150],[5,5])      ; create 4-bar at (300,150) to scale*5
 bard = Wireframe (barf, "pline")         ; draw the 4-bar as a polyline

 Lpta = locate (j4,origin,size)
 Lptb = locate (j5,origin,size)
 Lptc = locate (j6,origin,size)
 Lptd = locate (j7,origin,size)
  Lht = 5.0
  Lwd = 0.85
 Lrot = 0.0

labela = label (Lpta, Lht, "bar a")       ; label bars by putting label at the
labelb = label (Lptb, Lht, "bar b")       ; mid points of each of the bars and
labelc = label (Lptc, Lht, "bar c")       ; the base
labeld = label (Lptd, Lht, "bar d")


 theta = 0.2                              ; now cycle through different values
 theta = 0.4                              ; theta to create the locus
 theta = 0.6
 theta = 0.8                              : Later I found a better way to do this.
 theta = 1.0
 theta = 1.2
 theta = 1.4
 theta = 1.6
 theta = 1.8
 theta = 2.0
 theta = 2.2
 theta = 2.4
 theta = 2.6
```

Script 3. Parametric Drawing Frame

This script draws a drawing frame on any standard "A" size drawing sheets, orienting the sheets as landscape or portrait. A title-block is drawn in the bottom right hand corner with text inserted as per the labels in the definitions.

```
;; DRAWING FRAME  DRGFRAME.LSP
;; Drawing Frame for std drawings
;; by A.J.Cartwright
;; created 29 March 1993

LLreal : paper drg blk0 blk02 Dframe                ; Declare variables
LLreal : blk1 blk2 blk3 blk4 blk5 blk6 blk7
LLreal : blk1f blk2f blk3f blk4f blk5f blk6f blk7f
frame  : blk1d blk2d blk3d blk4d blk5d blk6d blk7d
frame  : frameD
Lreal  : paprBL paprTL paprTR paprBR
Lreal  : drgBL drgTR drgTL drgBR
Lreal  : blkSC blkTL blkBL blkTR blkBR

str    : orient
int    : Asize BL TL x y
real   : len height blkDx blkDy

Asize  = 1                                          ; 'A1'-size paper, default
orient = "landscape"                                ; orientation may be portriat
paper  = [paprBL,paprTL,paprTR,paprBR]              ; coords of paper corners
drg    = [drgBL, drgTL, drgTR, drgBR]               ; and drawing frame corners
blk0   = [blkBL, blkTL, blkTR]                      ; corner title block coords

len    = 1000*expt(2.0,0.25-Asize/2.0               ;Standard 'A'-size paper
height = len/sqrt(2.0)                              ; title block height
BL = 1                                              ; Bottom left. These act as
TL = 2                                              ; Top left      labels for projectn
x = 1
y = 2

paprBL = [0, 0]                                     ; paper bottom left coords
paprTR = if orient = "landscape" then [len, height] ; paper top-right coords
                                 else [height, len]
drgBL  = [len/40.0,len/40.0]                        ; drg frame bottom left
drgTR  = vdiff(paprTR, drgBL)                       ;  top left coords
drgTL  = [projn(x,drgBL), projn(y,drgTR)])          ;  bottom right
drgBR  = [projn(x,drgTR), projn(y,drgBL)])          ;  bottom right
blkSc  = vscale([0-0.1,0.025],log(Asize+3.0))       ;-ve to move origin to B

blkBR  = paprBL                                     ; BR based on (0, 0)
blkTL  = locate([len,height], paprBL, blksc)        ;  and fraction
blkTR  = [projn(x,blkBR), projn(y,blkTL)]           ;  of paper size
blkBL  = [projn(x,blkTL), projn(y,blkBR)]
blk02  = vshear(blk0, [2.0,2.0])

blkDx  = projn(x,blkTR)-projn(x,blkTL)              ; Dy and Dx
blkDy  = projn(y,blkTL)-projn(y,blkBL)              ;  displacements

blk1   = vtrans(blk0, drgBR)                        ; locate coords of
blk2   = vtrans(blk1, [0.0, blkDy])                 ;  each frame for inserting text
blk3   = vtrans(blk2, [0.0, blkDy])
blk4   = vtrans(blk1, [(0.0-blkDx), 0.0])
```

```
blk5   = vtrans(blk4, [0.0, blkDy])
blk6   = vtrans(blk5, [0.0, blkDy])
blk7   = vtrans(blk02, projn(3,blk3))

Dframe = cedge(drg)                                  ; create frames themselves
blk1f  = pedge(blk1)
blk2f  = pedge(blk2)
blk3f  = pedge(blk3)
blk4f  = pedge(blk4)
blk5f  = pedge(blk5)
blk6f  = pedge(blk6)
blk7f  = pedge(blk7)

frameD = Wireframe(Dframe, "pline")                  ; draw the lines demarking
blk1D  = Wireframe(blk1f, "pline")                   ;  the title block
blk2D  = Wireframe(blk2f, "pline")
blk3D  = Wireframe(blk3f, "pline")
blk4D  = Wireframe(blk4f, "pline")
blk5D  = Wireframe(blk5f, "pline")
blk6D  = Wireframe(blk6f, "pline")
blk7D  = Wireframe(blk7f, "pline")

; TEXT LABELS
str    : labl1 labl2 labl3 labl4 labl5 labl6 labl7   ; declare label variables
str    : name date title namet datet titlet Univ
lreal  : loc1 loc2 loc3 loc4 loc5 loc6 loc7
lreal  : offset1 offset2
real   : txtht

name   = "A J Cartwright"                            ; label text
date   = "March 1993"
title  = "TEST"
namet  = "NAME"
datet  = "DATE"
titlet = "TITLE"
Univ   = "UNIVERSITY OF WARWICK"

txtht  = BlkDy*0.4                                   ; Text height
offset2= [blkDx*0.1, blkDy*0.25]  ; [1.0,2.5]        ; start position within frame
offset1= vshear(offset2,[2.0,2.0])

loc1   = vtrans(projn(BL, blk1),offset2)             ; Actual positions of text labels
loc2   = vtrans(projn(BL, blk2),offset2)
loc3   = vtrans(projn(BL, blk3),offset2)
loc4   = vtrans(projn(BL, blk4),offset2)
loc5   = vtrans(projn(BL, blk5),offset2)
loc6   = vtrans(projn(BL, blk6),offset2)
loc7   = vtrans(projn(BL, blk7),offset1)

labl1  = label(loc1, txtht, name)                    ; functions to place text on
labl2  = label(loc2, txtht, date)                    ;  the display itself
labl3  = label(loc3, txtht, title)
labl4  = label(loc4, txtht, namet)
labl5  = label(loc5, txtht, datet)
labl6  = label(loc6, txtht, titlet)
labl7  = label(loc7, txtht, univ)
```

Script 4. Elastic Hinge Design

The formulae for an elastic notch type hinge are given and then applied to the design of a monolith, a lever system for amplifying the input displacement by a factor of about 50.

```
; Monolith.Lsp
; by A.J.Cartwright
; New version  5 Apr 93
; revision    20 Apr 93

real  : Th R Z L E Omax
real  : F Mmax K Kt Qmax
real  : thetam lamda

E = 205000.0                                  ; Young's Modulus - steel (MPa)
z = 10.0                                      ; hinge width (mm)
Th = 0.2                                      ; hinge thickness at smallest, mm
R = 2.0                                       ; radius of hinge, mm
L = 25.0                                      ; length of lever arm, mm
Omax = 250.0                                  ; max strength of material, MPa
F = 400.0                                     ; force on lever: N

K    = 0.166 + (0.565 * th / R)               ; stress concentration factor
Mmax = Z*th*th*Omax / (6*Kt)                  ; max bending moment
Kt   = 0.325 + ((2.7*th)+(5.4*R))/(8*R + th)) ; total geometry factor
thetam = 4*K*R*Omax / (Kt*E*th)               ; max angle of displacement, rad
lamda  = E*Z*th*th*th / (6*K*R*L*L)           ; stiffness of system N/mm
Qmax   = thetam*L                             ; max displacement of lever mm

;Topology of the basic monolith hinge is just a cuboid.
;Its shape is a particular realisation of the basic cuboid

Lreal : NL FL FR NR
real  : ENL EFL EFR ENR Extr                  ; block vertices
real  : OO                                    ; origin
real  : BX BY BZ
real  : HX HY HZ LX LY LZ

BX = 50.0 * R                                 ; block dimensions
BY = 20.0 * R                                 ; as a function of R
BZ = evalID(Z)

HX = 2*R+Th                                   ; hinge dimensions
HY = 2*R
HZ = evalID(Z)

LX = evalID(HX)                               ; lever dimensions
LY = evalID(L)
LZ = evalID(Z)

OO  = [0,0,0]                                 ; origin

NL  = [0,0,0]                                 ; Base :Near/Left
FL  = [0,1,0]                                 ; Far/Left
FR  = [1,1,0]                                 ; Far/Right
NR  = [1,0,0]                                 ; Near/Right
```

```
Extr= [0,0,1]                                    ; extrusion vector
                                                 ; Top : Extruded/Near/Left etc
EFL = vtrans(FL,Extr)
EFR = vtrans(FR,Extr)
ENR = vtrans(NR,Extr)

LLstr : cuboidV
frame : cuboid
frame : block1  Vlever  Hlever  Hhinge  Vhinge
frame : block1D VleverD HleverD HhingeD VhingeD
str   : blocksh horizH vertH

cuboidV = extrude (["NL", "FL", "FR", "NR"])
cuboid  = complex (cuboidV)

block1 = object (cuboid, OO, [BX,BY,BZ])        ; base block
Vlever = object (cuboid, OO, [LX,LY,LZ])        ; dummy vertical lever
Hlever = object (cuboid, OO, [LY,LX,LZ])        ; dummy horizontal lever
Vhinge = object (cuboid, OO, [HX,HY,HZ])        ; dummy horizontal hinge
Hhinge = object (cuboid, OO, [HY,HX,HZ])        ; dummy vertical hinge

blocksh = "POLYHEDRON"
HorizH  = "Hhinge"
VertH   = "Vhinge"

block1D = Wireframe (block1, blocksh)
HleverD = Wireframe (Hlever, blocksh)
VleverD = Wireframe (Vlever, blocksh)
HhingeD = Wireframe (Hhinge, horizh)
VhingeD = Wireframe (Vhinge, verth)

frame : Hhinge1 Hhinge1D                         ; Put in particular levers and hinges
frame : Hhinge2 Hhinge2D
frame : Hhinge3 Hhinge3D
frame : Hhinge4 Hhinge4D

frame : Hlever1 Hlever1D
frame : Hlever2 Hlever2D
frame : Vlever1 Vlever1D
Lreal : O1 O2 O3 O4 I

 I = [1,1,1]
O1 = [13.0,13.0,0]
O2 = Vtrans(O1,[HY,0,0])
O3 = Vtrans(O2,[L,0,0])
O4 = Vtrans(O3,[HY,0,0])

Hhinge1 = object (Hhinge, O1, I)                 ;lever1
Hlever1 = object (Hlever, O2, I)
Hhinge2 = object (Hhinge, O3, I)

Hhinge3 = object (Hhinge1, [0,20,0], I)          ;lever2
Hlever2 = object (Hlever1, [0,20,0], I)
Hhinge4 = object (Hhinge2, [0,20,0], I)

Vlever1 = object (cuboid, O4, [HX,20.0+HX,LZ])
                                                 ; Realise (display) objects in
```

```
                                                    ; appropriate surface shapes
Block1D  = Surface (block1, blocksh)                ; outer block

Hhinge1D = Surface (Hhinge1, HorizH)
Hlever1D = Surface (Hlever1, blocksh)
Hhinge2D = Surface (Hhinge2, HorizH)

Hhinge3D = Surface (Hhinge3, HorizH)
Hlever2D = Surface (Hlever2, blocksh)
Hhinge4D = Surface (Hhinge4, HorizH)

Vlever1D = Surface (Vlever1, blocksh)
```

Script 5. Graph Plotting

The function at the commencement of the listing is used to create the array of coordinates. It illustrates the complexity of functions that underlie EdenLisp. Such functions need to be designed to suit particular applications. Once done the definitions are easy to manipulate.

```
; GRAPH.LSP
; by AJ Cartwright
; October 1993
(defun mkgraf2 (Npts xrange yrange xyfn$
                    / mkgrh xylst x y fxy)          ; Associate function values
    (defun mkintL (N)                               ;   with list members
     (if (>= N 0)                                   ; This is a separate help function
                                                    ;    It makes a list of int (1 2 3 4
                                                    5 ...)
     (append (mkintL (1- N)) (list N))              ;   by recursion through 0.. N
    ))

    (defun mkgrh (xylst func)                       ; This is a separate help function
     (cond                                          ;   It creates a list
      ((null xylst) nil)                            ;  of y values calculated from
      (T                                            ;   inserting x-values into
       (setq x (caar xylst))                        ;   the analytical function "func"
       (cons (list x y (eval (read func)))          ;   and returning  (x,y) coords list
             (mkgrh (cdr xylst) func)               ; It is recursive through xylist
    ))))                                            ; end of help function

 (setq xylst                                        ; Function starts here
  (mapcar
  '(lambda (x)                                      ; replace integers by (x,y) coords
    (list (* (/ (float xrange) Npts) x)             ; = a set of x values in x-range
          (* (/ (float yrange) Npts) x)             ; = a set of y values in y-range
   ))                                               ; by using an initial set
   (mkintL Npts)                                    ; of integers obtained  here
 ))
 (setq fxy (catlex (statmt (lex xyfn$))))           ; hand over actual (x,y) coords
 (mapcar                                            ; to EdenLisp
  '(lambda (lst)                                    ; having once computed the y value
    (setq y (cadr lst))                             ; by passing the x values to the
    (mkgrh xylst fxy)                               ; help function from the
   ) xylst                                          ; list created by the setq xylst above
))
```

```
; EdenLisp Script
str    : func                                    ; declare variables
int    : Npts
real   : xrange yrange
lreal  : origin
frame  : gpts gline flist

Npts   = 50                                      ; Number of points on one graph
xrange = 40.0                                    ; range of x coords
yrange = 40.0                                    ; range of y coords
func   = "cos(sqrt(x^2 + y^2))"                  ; func = graph of analytic curve

flist  = mkgraf (Npts, xrange, yrange, func)     ; compute set of z coords
origin = [100.0,100.0,100.0]                     ; new origin
gpts   = object(flist, origin, [1,1,1])          ; move (x,y,z) set to new origin
gline  = wireframe (gpts, "carpet")              ; and display as a carpet graph
```

Script 6 Denture Design

Script 6*a*. 2D mouth profile and tooth form

The first part of this script generates the list of points that go to make up each of the archetypal tooth shapes. Actual dummy points are created in order to make it easy to edit. Actions are used to generate all the teeth required.

```
;  DENTAL2.LSP
;  by AJ Cartwright
;  11th November 1993
;  rev 1st Dec 1993

; Dental features
; UpperLeft, UpperRight, LowerLeft, LowerRight
6 or 8 e.g.  UL1 UR7 LL4 LR8
;   Tooth   condition:   N.natural,   M.missing,
A.artificial
; Tooth changes allowed  N->M, M->A only
; Types: incisor canine molar wisdom
; Artificial features: Saddle, pad=compressive
support, hook=tensile support

; 2D Topology of tooth
; C of G must be within the region and on an
arc
; consists of a closed polyedge of N points

; TOPOLOGY
str  :  Mkincisor  Mkcanine  Mkmolar  Mkwisdom
mkpad
llreal : incisor canine molar wisdom pad

                                                 ; Actions  A_lst
Mkincisor = A_lst ("incisor","i","lreal",10)     ; generates labels for
Mkcanine  = A_lst ("canine", "c","lreal",10)     ; tooth shape points
Mkmolar   = A_lst ("molar",  "m","lreal",16)     ; creating lists such as
Mkwisdom  = A_lst ("wisdom", "w","lreal",16)     ; (w1,w2,w3....w12)
Mkpad     = A_lst ("pad", "p", "lreal",12)
```

```
;;;  GEOMETRY
i1 = [ 17,  0]                ;    incisor tooth shape
i2 = [ 15,  9]
i3 = [  7, 12]                ;       4        3
i4 = [ -5, 13]                ;  5                  2
i5 = [-14,  7]                ; 6          +          1
i6 = [-17,  1]                ;  7                  10
i7 = [-13, -7]                ;     8            9
i8 = [ -7,-11]
i9 = [  2,-13]
i10= [ 13,-11]

c1 = [ 15,  0]
c2 = [ 17, 12]                ;      canine tooth shape
c3 = [  5, 13]
c4 = [ -5, 13]                ;          4      3
c5 = [-17, 12]                ; 5                        2
c6 = [-15,  0]                ;   6          +         1
c7 = [-10, -8]                ;     7                10
c8 = [ -6,-13]                ;          8       9
c9 = [  6,-13]
c10= [ 10, -8]

m1 =  [ 20,  0]               ;          molar shape
m2 =  [ 19,  7]
m3 =  [ 15, 11]
m4 =  [ 10, 15]               ;   6         5          4
m5 =  [  0, 14]               ;  7                        3
m6 =  [-10, 15]               ; 8                          2
m7 =  [-15, 11]               ; 9          +               1
m8 =  [-19,  7]               ; 10                        16
m9 =  [-20,  0]               ;   11                            15
m10 = [-19, -7]               ;      12         13         14
m11 = [-15,-11]
m12 = [-10,-15]
m13 = [  0,-14]
m14 = [ 10,-15]
m15 = [ 15,-11]
m16 = [ 19, -7]

w1 =  [ 21,  0]               ;          wisdom tooth shape
w2 =  [ 20,  8]
w3 =  [ 16, 12]
w4 =  [ 11, 16]               ;    6           5          4
w5 =  [  0, 15]               ;  7                           3
w6 =  [-11, 16]               ;  8                           2
w7 =  [-16, 12]               ;  9           +               1
w8 =  [-20,  8]               ;  10                         16
w9 =  [-21,  0]               ;    11                                15
w10 = [-20, -8]               ;       12          13          14
w11 = [-16,-12]
w12 = [-11,-16]
w13 = [  0,-15]
w14 = [ 11,-16]
w15 = [ 16,-12]
w16 = [ 20, -8]
```

```
p1 =  [ 5.0, 0.0]                    ;     artificial pad shape
p2 =  [ 4.0, 3.0]
p3 =  [ 1.0, 4.0]
p4 =  [ 0.0, 5.0]
p5 =  [-1.0, 4.0]
p6 =  [-4.0, 3.0]
p7 =  [-5.0, 0.0]
p8 =  [-4.0,-3.0]
p9 =  [-1.0,-4.0]
p10 = [ 0.0,-5.0]
p11 = [ 1.0,-4.0]
p12 = [ 4.0,-3.0]

; Dental arrangement of mouth
real    : pi rt Rx Ry ang Z ScM                    ; declare variables used below
lreal   : scaleA
real     : ang1 ang2 ang3 ang4 ang5 ang6 ang7      ; scale used in display
ang8
real    : mouth male female child Nteeth           ; type of tooth

male    = 1.0                                      ; Scale for mouth size
female = 0.9
child  = 0.7
Nteeth = if ScM=0.7 then 6.0 else 8.0
pi     = 3.14159265
rt     = pi/2.0

mouth  = male                                      ; define current mouth shape
ScM    = evalid(mouth)                             ; use its real value
Rx     = 200.0*ScM                                 ; elliptical shape of mouth minor
                                                   axis
Ry     = 250.0*ScM                                 ; elliptical shape of mouth major
                                                   axis
Z      = 1.0                                       ; Z scale (if 3D added)
SCALEa = if ScM=0.7 then [0.9,0.9,1.0]             ; scale Rxy
                    else [ScM,ScM,1.0]

Nteeth = if ScM=0.7 then 6.0 else 8.0              ; child has no wisdom teeth
ang    = pi/Nteeth*2.24                            ; position increment round ellipse

ang1   = (Nteeth + 0.2)*ang                        ; actual position of first tooth
ang2   = (Nteeth - 0.8)*ang                        ; and 2nd tooth, etc
ang3   = (Nteeth - 1.8)*ang
ang4   = (Nteeth - 2.8)*ang
ang5   = (Nteeth - 3.9)*ang
ang6   = (Nteeth - 5.0)*ang
ang7   = (Nteeth - 6.1)*ang
ang8   = (Nteeth - 7.2)*ang

str       : tooth  toothp  natural  artificial  ; declare strings
missing
natural = "cpline"                                 ; tooth is shown as a polyline
artificial = "fill"                                ; artificial tooth is shown hatched
tooth   = natural                                  ; define current tooth type
toothp  = if ScM=0.7 then "" else tooth


str  : URtd URpd URid URp URt UR1 URd
Lstr : UR UR1 UR2 UR3 UR4 UR5 UR6 UR7 UR8
```

```
URtd= "llreal: ?1?2typ "                              ; Dummy text used with actions
URpd= "lreal : ?1?2pos "                              ;    to create the definitions
URid= "frame : ?1?2ins ?1?2dsp"                       ;    for each tooth shape.
URp     =     "?1?2pos     =     [Rx*cos(ang?2),      ; Middle of tooth profile
Ry*sin(ang?2)]"
URt = "?1?2typ = rotobj(?3, rt-ang?2,  Z)"            ; orientation & type of tooth
URi   =  "?1?2ins  =  object(?1?2typ,  ?1?2pos,       ; instance of tooth
scalea)"
URd = "?1?2dsp =  wireframe(?1?2ins,?4)"              ; display the profile

UR  = [URtd, URpd, URid, URp, URt, URi, URd]

UR1=A_repl(["UR","1","incisor","tooth"], UR)
UR2=A_repl(["UR","2","incisor","tooth"], UR)
UR3=A_repl(["UR","3","canine","tooth"], UR)
UR4=A_repl(["UR","4","molar","tooth"], UR)
UR5=A_repl(["UR","5","molar","tooth"], UR)
UR6=A_repl(["UR","6","molar","tooth"], UR)
UR7=A_repl(["UR","7","wisdom","toothp"], UR)
UR8=A_repl(["UR","8","wisdom","toothp"], UR)

str  : ULp ULt
Lstr : UL UL1 UL2 UL3 UL4 UL5 UL6 UL7 UL8

ULp = "?1?2pos = Vshear(UR?2pos, [-1,1])"
ULt = "?1?2typ = rotobj(?3, ang?2-rt,  Z)"
UL  = [URtd, URpd, URid, ULp, ULt, URi, URd]

UL1=A_repl(["UL","1","incisor","tooth"], UL)
UL2=A_repl(["UL","2","incisor","tooth"], UL)
UL3=A_repl(["UL","3","canine","tooth"], UL)
UL4=A_repl(["UL","4","molar","tooth"], UL)
UL5=A_repl(["UL","5","molar","tooth"], UL)
UL6=A_repl(["UL","6","molar","tooth"], UL)
UL7=A_repl(["UL","7","wisdom","toothp"], UL)
UL8=A_repl(["UL","8","wisdom","toothp"], UL)

str  : LLp LLi
Lstr : LL LL1 LL2 LL3 LL4 LL5 LL6 LL7 LL8

LLp="?1?2pos = Vshear(UR?2pos, [1,-1])"
LLi="?1?2ins    =    object(UL?2typ,    ?1?2pos,
scaleA)"

LL = [URtd, URpd, URid, LLp, LLi, URd]

LL1=A_repl(["LL","1","incisor","tooth"], LL)
LL2=A_repl(["LL","2","incisor","tooth"], LL)
LL3=A_repl(["LL","3","canine","tooth"], LL)
LL4=A_repl(["LL","4","molar","tooth"], LL)
LL5=A_repl(["LL","5","molar","tooth"], LL)
LL6=A_repl(["LL","6","molar","tooth"], LL)
LL7=A_repl(["LL","7","wisdom","toothp"], LL)
LL8=A_repl(["LL","8","wisdom","toothp"], LL)

str  : LRp LRi
Lstr : LR LR1 LR2 LR3 LR4 LR5 LR6 LR7 LR8
```

```
LRp="?1?2pos = Vshear(UR?2pos, [-1,-1])"
LRi="?1?2ins    =    object(UR?2typ,    ?1?2pos,
scaleA)"

LR = [URtd, URpd, URid, LRp, LRi, URd]
LR1=A_repl(["LR","1","incisor","tooth"], LR)
LR2=A_repl(["LR","2","incisor","tooth"], LR)
LR3=A_repl(["LR","3","canine","tooth"], LR)
LR4=A_repl(["LR","4","molar","tooth"], LR)
LR5=A_repl(["LR","5","molar","tooth"], LR)
LR6=A_repl(["LR","6","molar","tooth"], LR)
LR7=A_repl(["LR","7","wisdom","toothp"], LR)
LR8=A_repl(["LR","8","wisdom","toothp"], LR)

llreal : PADtyp
frame : PADins PADdsp
padtyp = rotobj(pad, rt-angl,  Z)
padins = object(PADtyp, UR1pos, scalea)
paddsp = wireframe(PADins,artificial)
```

Script 6*b* Denture Plate

The following script for the generation of the dental plate needs to be run after the first script as it makes use of the definitions there.

```
;  DENTAL3.LSP
;  Plate design
;  by AJ Cartwright
;  1st Dec 1993

str   : t1 mkplat
lstr  : mkp12 mkp45 mkp23 mkp67
llreal : plate
frame : plateO plateD
lreal : UR23pl UR45pl UL23pl UL45pl

t1    =    "?1?2?3pl    =    midpt(?1?2pos,
?1?3pos)"
mkp12  = A_repl (["UR","2","3"], [t1])      ; Position of gaps
mkp45  = A_repl (["UR","4","5"], [t1])      ; over which denture
mkp23  = A_repl (["UL","2","3"], [t1])      ; is to act
mkp67  = A_repl (["UL","4","5"], [t1])
mkplat = A_lst ("plate","pl","lreal",28)    ; create list of 28 labels

pl1=projn(7,UR3ins)                         ; profile of denture
pl2=projn(6,UR3ins)                         ; takes values from
pl3=projn(5,UR3ins)                         ; the appropriate
pl4=projn(4,UR3ins)                         ; tooth forms
pl5=projn(3,UR3ins)
pl6=projn(2,UR3ins)

pl7=projn(6,UR4ins)                         ; and joins the teeth
pl8=projn(5,UR4ins)                         ; together
pl9=projn(4,UR4ins)
pl10=projn(3,UR4ins)
pl11=projn(2,UR4ins)
```

```
pl12=projn(1,UR4ins)
pl13=projn(16,UR4ins)
pl14=projn(15,UR4ins)

pl15=projn(11,UL4ins)                    ; crosses over to the
pl16=projn(10,UL4ins)                    ; opposite quarter of
pl17=projn(9,UL4ins)                     ; the mouth
pl18=projn(8,UL4ins)
pl19=projn(7,UL4ins)
pl20=projn(6,UL4ins)
pl21=projn(5,UL4ins)
pl22=projn(4,UL4ins)

pl23=projn(5,UL3ins)
pl24=projn(4,UL3ins)
pl25=projn(3,UL3ins)
pl26=projn(2,UL3ins)
pl27=projn(1,UL3ins)
pl28=projn(10,UL3ins)

plateO = object(plate, [0,0,0], scaleA)  ; make the object
plateD = wireframe(plateO, natural)      ; display as polyline

; TEXT LABELS
real   : txtht
lreal  : offset ofset2

txtht  = 10.0
offset = [0, 20.0]
ofset2 = [-55.0,5]

str    : labl1 labl2 labl3 labl4
lreal  : loc1 loc2 loc3 loc4
loc1   = vtrans(projn(1, UR3ins),offset)  ; position the labels
loc2   = vtrans(projn(1, UR4ins),offset)
loc3   = vtrans(projn(1, UL3ins),ofset2)
loc4   = vtrans(projn(1, UL4ins),ofset2)

labl1  = label(loc1, txtht, "UR3")        ; label tooth "UR3"
labl2  = label(loc2, txtht, "UR4")
labl3  = label(loc3, txtht, "UL3")
labl4  = label(loc4, txtht, "UL4")
```

Published Papers Not filmed for Copyright reasons

4

DX

230524